CMU-CS-82-137

(12)

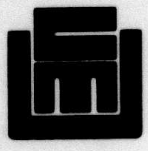
Tailoring Software for Multiple Processor Systems

Karsten Schwans

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

1 October 1982

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

CMU-CS-82-137

Tailoring Software for Multiple Processor Systems

Karsten Schwans

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

1 October 1982

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

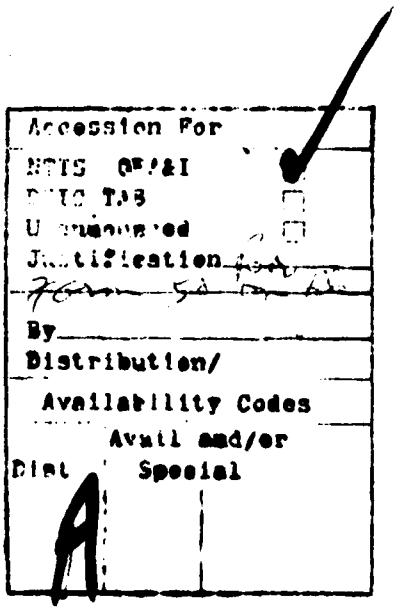
Abstract

Multiple processor systems are becoming increasingly common. However, their use remains difficult due to a lack of knowledge concerning the development of parallel application programs. In addition, contrary to popular predictions of "cheap and plenty" resources, efficient management of distributed processor and memory resources remains of critical importance to the successful use of these systems.

The contributions of this thesis are twofold. First, we design and implement a programming environment for multiple processor applications, called the TASK system. Second, we discuss the integration of policies and mechanisms for resource management into the TASK system.

In TASK, application programs are written in terms of the abstractions offered by the operating system used for program execution. As a result, once an application program is written, its execution requires few additional efforts by the application's programmer. Programs are written in two languages. The TASK language, designed and implemented as part of this thesis, is used to describe the logical structure of an application program, and an existing, algorithmic language is employed to implement the application's algorithms. The construction of an executable version of an application from the TASK and algorithmic language programs is automated. Such construction includes automatic linking and loading as well as the automatic allocation of resources to the individual components of the application program.

Programmers guide the allocation of hardware resources to program components by stating high-level directives in TASK programs. To identify suitable directives and to develop procedures that automatically perform resource allocation based on these directives, we develop a model of multiple processor software and hardware, called the proximity model. The model, the directives, and the resource allocation procedures are tested by experimentation with application programs on the Cm* multiprocessor.



Acknowledgements

This thesis involves the help of many members of the Computer Science Department at Carnegie-Mellon University. Interactions with the STAROS group have helped shape the TASK system into a practical, useful tool for the user community of Cm*. Special thanks go to Mike Kazar for writing the loader that is part of this system. Ed Gehringer and Robert Chansler deserve credit for the TASK language manual, Jarek Deminet performed the experiments that were the basis for the development of TASK's, and Ivor Durham assisted in the implementation of a suitable TASK user interface. In addition, I gratefully acknowledge Prof. Cheryl Gaimon's (formerly of Carnegie-Mellon's Graduate School of Industrial Administration) technical assistance in the formulation of a mathematical model for resource allocation and in the development of automatic resource allocation procedures.

My thesis committee, Profs. Anita Jones, Nico Habermann, Richard Rashid, Don Thomas, and William Wulf, contributed many knowledgeable and challenging comments throughout the conception and completion of this thesis. Furthermore, Anita Jones, Cheryl Gaimon, and Nico Habermann each devoted considerable time and support to translating this thesis into English. However, foremost, my thesis chairperson, Prof. Anita Jones, has earned my sincere gratitude. She first conceptualized the need for the TASK system. The TASK language design and the design of the language concepts that are relevant to automatic resource allocation were the result of many challenging hours of joint work. Her patient and insistent guidance during system implementation and experimentation have had a major impact on the quality of this research.

Table of Contents

List of Figures

List of Tables

List of Examples

1. Introduction

1.1. Problem Statement and Thesis Goals

While multiple processor hardware is becoming increasingly common, the current level of software support is insufficient. Few programming tools exist and existing tools are difficult to use [29, 84, 166]. Furthermore, although theoretical research has been performed in the areas of parallel scheduling and distributed data management [18, 150, 27], such research has had little practical impact. While programmers are provided with elementary mechanisms for resource management, they are given few practical aids in making appropriate resource management decisions.

Despite the lack of programming support, the use of multiple processor systems has grown substantially. Software has been developed to capitalize on the enhanced reliability or cost-effective performance of multiple processor systems [17, 91, 29, 106, 31, 35], and software that executes on physically distributed systems has been written [98, 16, 51, 152, 59, 148, 120]. Therefore, programming support in addition to what is available today is long overdue.

Based on our experiences with multiple processor systems [28, 29, 166], we have designed and implemented a programming environment [62, 113, 127] for multiple processor application programs called the TASK system. To focus on multiple processor systems, the design and implementation of TASK is restricted. Specifically, TASK is constructed as a *tool system* [156] into which several program construction tools are integrated, namely compilers, linkers, and loaders. Furthermore, the TASK tool system has deliberately been built to support single programmers in the construction of large application programs. Therefore, the complexities caused by programmer teams cooperatively constructing large software systems are not addressed [70].

Since our experiences with multiprocessors indicate that resource management in multiple processor systems is difficult while the benefits attained from it are substantial [29, 28, 166, 84, 111], the

TASK system assists programmers in making resource management decisions. Specifically, programmers need not allocate specific hardware resources to individual program components. Instead, such allocation decisions are automatically made based on high-level *resource directives* stated by application programmers, where each directive expresses allocation preferences or constraints concerning a set of program components. Therefore, resource allocation is partially automated in TASK, and programmers need not explicitly deal with the large number of different components in software and in distributed hardware of substantial size.

Resource directives are designed and processed based on a general model of software and hardware (see Chapter 3). As a result, our methods of partially automating resource allocation can be applied to a variety of multiple processor software and hardware. However, resource allocation in TASK is implemented for the special cases of the Cm* multiprocessor [57] and the STAROS operating system [80].

1.2. Software for Multiple Processor Systems

Multiple processor systems range from loosely coupled, local or distributed networks [64, 126, 115], to tightly coupled multiprocessors [57, 167], to multi-ALU systems consisting of synchronously executing processor and memory units [155, 54, 104, 87, 3]. Multi-ALU systems are not considered in this thesis. Instead, processor and memory units are assumed to execute independently of and asynchronously to other processors in the system [57, 167]. However, whether a multiple processor system is designed to perform a specialized task [136, 94, 17, 110, 65] or whether it serves as a basis for a variety of application programs [148, 120, 167, 57, 124, 6] is not of concern to our investigations.

Multiple processor application programs differ from uniprocessor software in two ways. First, parallelism is exploited explicitly in multiple processor applications, often with the intent of realizing the potentials of increased cost-performance or reliability exhibited by multiple processor systems [51, 65, 96, 17, 84]. However, realizing these potentials depends upon resource management

in which the performance-related effects of using large numbers of distributed resources are accurately predicted. Since such predictions are not possible given our current knowledge of multiple processor systems [29, 84, 81, 27], the second difference between multiple processor and uniprocessor software is that resource management in multiple processor systems cannot be transparent to programmers.

Research efforts during the last decade acknowledge the differences between uniprocessor and multiple processor software mentioned above. To exploit parallelism, new programming languages [50, 68, 14, 15, 71, 33, 137] have been designed, and existing languages have been extended to provide mechanisms for concurrent programming [116, 67, 13, 102]. Furthermore, new operating systems provide mechanisms for resource management and mechanisms that facilitate cooperation and communication between independently executing processes of concurrent programs [80, 116, 102, 128, 147, 24, 67, 134]. However, due to the difficulty of resource management, resource allocation is done at the "assembly level" in these languages and operating systems--programmers must allocate specific resources to individual software components. Therefore, programmers do not receive any support in dealing with common characteristics of multiple processor systems [98, 51, 136, 3, 57, 64, 136, 59] such as inhomogeneous or asymmetric resources [84, 108], inhomogeneous access to available resources, and a high probability of dynamic configuration change [141].

In TASK, application programs are described as *task forces*--multiple processes that cooperate and communicate to achieve a common goal [79]. A task force is specified with the TASK language [81], whose design and implementation are discussed in this thesis. TASK programs describe the components of a task force, including its processes and the components that contain the processes' code and data. However, the code executed by each process is written in a standard, algorithmic programming language, the Bliss-11 language [161]. TASK programs contain sufficient information so that the Bliss programs in each task force can be automatically linked and the task force can be automatically loaded. The relationships between the TASK and Bliss compilers, the linker, and the loader are

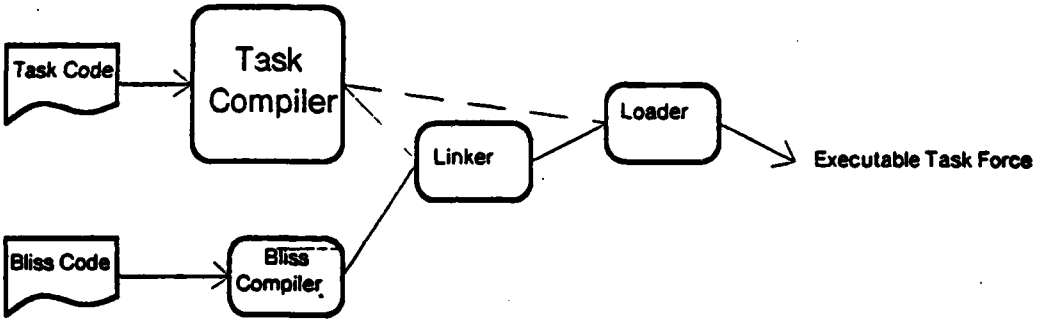


Figure 1-1: Compiling a Set of BLISS Programs and a TASK program

depicted in figure1-1. Note that the output of the TASK compiler is used to drive the linker and loader (see the dashed lines). The result of linking and loading is an executable form of the task force.

The TASK system recognizes the difficulty of resource management by automatically allocating specific resources to individual task force components given programmer-defined resource directives. Such directives are stated in TASK programs. Given a directive expressing preferences or constraints concerning the allocation of resources to a set of task force components, TASK makes allocation decisions based on its knowledge of the hardware and on its current allocation policy. In figure(1-2), we display the different items of information used for resource allocation.

Programmers stating resource directives are not required to know the current hardware configuration used for task force execution. The resulting hardware transparency enables programmers to write task forces that can execute on any hardware configuration containing sufficient resources. Furthermore, since programmers need not know the policies used by TASK for resource allocation, intimate knowledge of the performance characteristics of the distributed hardware is not required in

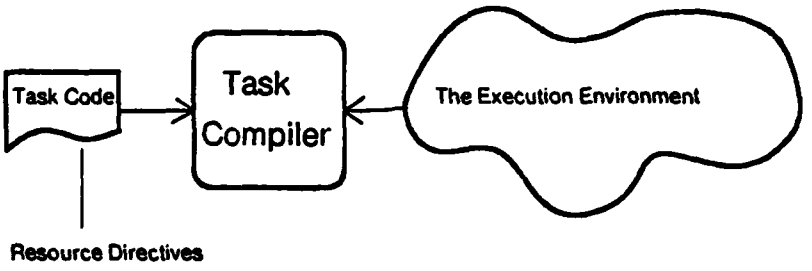


Figure 1-2: The Sources of the Information used by TASK

order to run a task force implemented within the TASK system.

The TASK tool system is a programming environment for the development of task forces. However, to focus on resource management, we do not deal with the following issues addressed in programming environments related to TASK: module interface control, system version control [156, 62], and interactions with editing or debugging tools [62]. Instead, TASK supports the specification and automatic construction of task forces, and task forces are executed under the control of the STAROS operating system [80]. A single-process debugger is available [62, 109], and a task force monitor [144] has been partially integrated into the TASK tool system. The TASK system is used as part of an ongoing research effort [29].

1.3. Software Tailoring

Experiments with Cmmp and Cm* [29, 84, 166] have shown that the performance of an executing task force strongly depends on the allocation of resources to its components--we say that a task force must be *tailored* to its execution environment. The objective of tailoring is to execute a task force "efficiently" on different configurations of execution environments, where the metrics used to measure "efficiency" vary. Configurations differ in the numbers of available hardware components,

the amounts of available memory, and the available specialized hardware such as components with attached disks or network links. In addition, configurations can differ with respect to the topology of the interconnection network providing access to non-local processor and memory units.

We investigate the manner in which tailoring can be performed during task force construction, where tailoring is assumed to leave task force structure and algorithmic code unchanged [81]. An alternative to tailoring during task force construction (*static* tailoring) is tailoring during task force execution (*dynamic* tailoring). While static tailoring is concerned with constructing executable task forces that are well-suited to their execution environments, dynamic tailoring is primarily concerned with adapting executing software to changes in workload [38] or in usage patterns of distributed hardware [146]). In this thesis, emphasis is on static tailoring, but we will show that the methods used to automate static tailoring can also be applied to dynamic tailoring.

In the remainder of this section, the importance of task force tailoring is demonstrated in examples of tailoring for uniprocessor and multiprocessor systems. Furthermore, the difficulty of tailoring is discussed. The section concludes with an outline of task force tailoring in the TASK system.

1.3.1. Traditional Tailoring Examples

Von Neumann machines have the following limitations: small numbers of registers, small primary memory, and a hierarchy in access times to primary, secondary, and tertiary memory. Significant efforts have been made to overcome these limitations. For example, compilers must optimize the allocation of fast registers to subexpressions within the algorithmic code, or compilers must reorganize code to increase register usage [99, 92]. A comparative study of several compilers demonstrates the drastic effects of such optimization on the speed of program execution [49]. Furthermore, based on analytical research and practical experimentation concerning the memory reference patterns of application programs [76, 52, 38], automatic paging procedures have been implemented that "tailor" programs during execution. In addition, programs are reorganized during linking and loading such that their accesses to code and data exhibit a high locality of reference [52]. Sig-

nificant improvements have been derived from such static or dynamic "tailoring" [76].

1.3.2. Tailoring in Multiple Processor Systems

The use of uniprocessors as timesharing or multiprogramming systems is possible partly due to static and dynamic "tailoring" as described in the previous subsection. As is demonstrated below, tailoring is also of critical importance for many applications of tightly or loosely coupled multiple processor machines [29, 28, 166, 84, 111]. We consider the importance of tailoring in both small grain [67, 89, 39, 41] and large grain parallelism [50, 68, 14, 15, 71, 102, 33].

Small grain parallelism exists at the level of individual instructions in a program. At this level, tailoring is accomplished by automatically decomposing programs into small groups of machine instructions or partial instructions [54] that can be executed in parallel. The intent of decomposition is to maximize parallelism and therefore, to minimize a program's execution time. With respect to vector and array processors [104, 155, 54], automatic program decomposition has capitalized on the ease with which data arrays or vectors can be partitioned into separately processable parts [91, 88]. Experimental systems performing automatic parallel decomposition have been built and tested on tightly coupled multiple processors [67] and on simulators [89]. In both cases, automatic tailoring has successfully generated small amounts of parallelism. However, parallelism is limited by array dimensions. In addition, control and data must be distributed to available computing cells [3, 88] in order to minimize the time spent making data accessible to cells [3, 88, 91]--information must be routed to the right place at the right time.

Tailoring at the level of small grain parallelism has achieved recent importance in VLSI research, where decomposable, highly parallel, synchronous algorithms are being developed [90]. Similarly, tailoring is of importance at this level in the design and implementation of languages for data flow machines [39]. In the latter case, programs are represented so that parallelism in program data and control flow is easily recognized, and compilers, interpreters, and runtime systems are designed to capitalize on such parallelism [39].

While small grain parallelism can often be generated automatically, parallelism in the form of explicitly specified processes cannot. We call such parallelism *large grain parallelism*. As an example of tailoring at the level of large grain parallelism, consider processes that generate update or read requests to records of a distributed data base. A static tailoring problem can be phrased as follows: at which network sites should multiple copies of data base records be located to maximize the speed of both read and update requests for those records? A number of known solutions to this tailoring problem rely on allocating memory in network sites such that the total cost of communication is minimized, thereby minimizing the use of the "network" resource [18]. A corresponding dynamic tailoring problem can be phrased as the following decision problem: to which copy of a record should a request be sent in order to minimize request duration or to optimize a global good, such as total system throughput [139]? As with task force tailoring in Cm*, tailoring in distributed data bases can dramatically affect performance.

Static and dynamic tailoring at the level of large grain parallelism were also performed during the implementation of the ArpaNet's message servers [86]. In this case, static tailoring determined the appropriate message sizes for various types of traffic patterns with the objective of avoiding *contention* in the network. Dynamic tailoring consisted of message routing within the network, such that both fast and reliable message transfer could be guaranteed. As with tailoring in distributed data bases, solutions to the tailoring problems in the ArpaNet tend to minimize the use of the "network" resource.

Further tailoring examples exist in real-time processing, where tailoring not only concerns delivering data "to the right place at the right time", but also concerns balancing processor workloads to adhere to predetermined execution schedules [61, 34]. In Chapter 3, such tailoring is discussed with respect to the Cm* multiprocessor.

To conclude, for vector and array computers [155, 104, 54] which support small grain parallelism, sophisticated algorithms are required to fetch information required by primitive operations prior to its

use. Otherwise, memory reference time costs may severely degrade parallelism. Typically, specialized hardware and programming languages are used to implement such algorithms [39]. For multiprocessors and networks which support large grain parallelism, the objects referenced by an executing processor (e.g. data or code components) must be located "near" the processor when it references them [84]. In both small and large grain parallelism, information is "near" when it is rapidly accessible.

1.3.3. The Difficulty of Tailoring

While tailoring is of critical importance in a wide range of multiple processor applications, tailoring decisions are not easily made. As with other well known problems, mathematical formulations of tailoring can be shown to be NP-complete (see Chapter 5). Three additional factors contribute to the difficulty of tailoring. From an application programmer's point of view, one factor contributing to the difficulty of tailoring is the large number and the diversity of software and hardware resources involved in the tailoring process. For example, when tailoring a small application program for a typical configuration of the Cm* system, 50 data components accessed by several processes [42] must be allocated to the memory and processor resources of 20 different machines.

A second factor contributing to the difficulty of tailoring is the lack of information and knowledge held by application programmers. For example, programmers may not understand the relationships among the following: the software parameters changed for tailoring purposes, the objective of tailoring, and the effects of tailoring at execution time. An example of a task force executing on Cm* illustrates this point. Consider an increase of the task force parameter "number of processes" which is initiated to reduce the execution time of the task force. If this increase in parallelism causes an inordinate increase in processor communication, then the stated objective is not achieved. Instead of reducing execution time, contention within Cm* causes the execution time of the larger task force to exceed the execution time of the smaller task force [29]. Similar complexities have been observed for other tailoring objectives on architectures other than Cm* [18, 27, 56].

A third factor contributing to the difficulty of tailoring is the variety of tailoring objectives pursued by programmers. As a result, although common tailoring objectives can be anticipated by designers of automatic tailoring procedures (see Chapter 5), these procedures cannot be of sufficient generality to support the various tailoring objectives for a spectrum of multiple processor architectures.

1.3.4. Tailoring in the TASK System

Multiple processor architectures, namely networks and multiprocessors, cannot be successfully employed until task force tailoring has become routine and well understood. Recent experiences [29, 166, 148, 84] suggest that the partial automation of tailoring based on programmer assistance can be implemented successfully.

In the TASK system, programmer assistance consists of specifying resource directives, and partial automation consists of using these directives in resource allocation. Each of the three factors contributing to the difficulty of tailoring is addressed. First, since TASK's resource directives are stated for sets of task force and hardware components, large and small numbers of components can be handled with comparable ease. Second, TASK's knowledge supplements a programmer's information concerning distributed hardware and appropriate tailoring strategies. Third, programmers can choose among several, different, built-in tailoring objectives and policies, and new objectives and policies are easily added to TASK.

The investigation of tailoring in TASK is a case study of the manner in which tailoring can be embedded into any multiple processor programming environment. Specifically, the information required for tailoring is identified, and its representation in TASK is discussed. The feasibility of automatic tailoring is demonstrated by implementing and evaluating several tailoring procedures, which are shown useful, albeit not equally suitable, for several, different multiple processor architectures, application programs, and tailoring objectives.

1.4. Main Contributions and Related Work

This thesis touches upon three areas of Computer Science: programming languages, software engineering, and operating systems. To the area of programming languages, we contribute the design and implementation of a software specification language for multiple processor applications [81] called the TASK language. A TASK program specifies a task force and contains sufficient information to control the use of linkers and loaders. Furthermore, TASK caters to the unique requirements of the Cm* architecture [57] and the STAROS operating system [80]. Resource directives in TASK programs are related to specific allocation decisions as path expressions are related to LOCK or P and V operations. As with path expressions, the implementation of resource directives requires that considerable intelligence be embedded into the TASK compiler. Specifically, while uniprocessor compilers must optimize register allocation, TASK must optimize the allocation of processor and memory units to task force components.

To the area of software engineering, we contribute the design and implementation of the TASK tool system, which can be used to develop software for multiple processor execution environments. Since the TASK system contains explicit representations of the task forces being developed, it can be used as a testbed for experimentation concerning task force tailoring. Examples of straightforward experimentation are the modification or exchange of tailoring procedures and the modification of information used for tailoring.

In the area of operating systems, we demonstrate the practicality and feasibility of a specific aspect of resource management, namely task force tailoring. In addition, we investigate the integration of the TASK system with an operating system in order to perform tailoring both statically and dynamically. Automatic tailoring is performed by making practical use of the models and optimization procedures of theoretical investigations [18]. However, the straightforward adaptations of the optimal tailoring algorithms discussed in the theoretical literature are too cumbersome and time-consuming [47] to be used within the TASK system. Instead, TASK's heuristic tailoring procedures [107] derive non-optimal

resource allocation decisions, which are shown to be in accordance with our expectations given the current knowledge of Cm*.

There are few other projects with goals similar to those of this thesis. Most related projects are either centered around particular languages [102, 15, 116, 50], or are concerned with specific multiple processor architectures or operating systems [88, 39, 80, 24, 147, 134]. Notable exceptions are the Spice and Cedar projects at Carnegie-Mellon University and at Xerox Parc respectively [148, 120], the Sara project at UCLA [45], the Roscoe and StarMod efforts at the University of Wisconsin [147, 33], and the effort at Amherst [97]. Each of these projects concerns the construction of integrated programming environments for large multiple processor systems.

The Spice project at CMU is aimed at constructing a programming and working environment for scientific research on a network of small computers. The Cedar project at Xerox Parc focuses on the construction of a programming environment that is primarily applied in the office of the future. The goals of the Roscoe and StarMod efforts at the University of Wisconsin are to investigate distributed processing in more generality [147, 33]. At this early project stage, however, we cannot precisely characterize project contents. The distributed processing group at the University of Amherst addressed research issues similar to those of TASK and STAROS [97]. However, since the original proposal, no further research results have been reported. The Sara project at UCLA is, perhaps, most akin in some of its research content to the TASK effort [45]. The design system for hardware in the Sara project was extended to investigate issues of distributed software design and implementation. A software specification language based upon a graph representation was proposed and implemented [8]. One marked difference between the Sara and the TASK systems is that TASK is only concerned with programming in-the-large [40], whereas the Sara system contains facilities to perform both programming in-the-large and in-the-small.

1.5. Thesis Overview

In Chapter 2, the TASK programming environment is described. A sample task force is used to present the specification of a task force in the TASK language. In addition, the interactions among compilers, linkers, and loaders in the construction of an executable version of a task force are explained. In Chapter 3, a model of distributed hardware and parallel software is developed. Based on this model, the objectives and metrics of task force tailoring are stated. In Chapter 4, the implementation of tailoring within the TASK programming environment is discussed. The heuristic, automatic tailoring procedures for Cm* are presented in Chapter 5. A summary of results and several possible extensions of this research are described in Chapter 6. Details of the TASK language and loader are described in four appendices.

2. Task Force Development

2.1. Overview

Task force development consists of the specification of a task force with programming languages and the automatic construction of an executable task force from these specifications. In TASK, each task force is programmed in two languages to separate the specification of the logical task force structure in-the-large [40] from the specification of the algorithms in-the-small. The logical structure of a task force is programmed in the TASK language. A TASK program describes a task force as a collection of modules. Each module is a unit of abstraction in the sense of Parnas [130]. The programs constituting the body of a module are written in the algorithmic language Bliss-11 [161]. These programs will be referred to as algorithmic code. They describe the algorithms executed by task force processes, the detailed implementation of data structures, and the control flow among processes.

The task force specified in a TASK program is constructed using the standard compile, link, and load sequence. Compilation of TASK and Bliss programs is performed by their respective compilers. The TASK compiler outputs instructions concerning linking and loading. Given these instructions, the linker performs two kinds of actions. First, the separately compiled Bliss programs in an individual module are linked to each other. Second, the linked modules and the loading instructions of the entire task force are formatted to load the task force one module at a time. The TASK loader transfers the modules from the compilers' and linker's host computer (a PDP-10) to the site of task force execution (Cm*) across a local Ethernet network. Each module in the task force is individually transferred and loaded onto the distributed hardware. The task force executes under control of the STAROS operating system [80].

In the TASK tool system, the tedious, error-prone job of task force construction is automated such that the details of the linker, loader, and operating system remain transparent to programmers. As a

result, small or large task forces can be constructed with comparable ease. However, TASK is not as comfortable a programming environment as one might desire. First, because version control mechanisms are not available [156. 62], TASK cannot guarantee that a constructed task force reflects the recent updates to its TASK or Bliss programs. Second, because module interfaces are not controlled after task force construction [122, 62], the algorithmic code must explicitly perform parameter checking. Third, since an executing task force can be changed [142, 111] by explicit invocation of operating system functions, TASK does not impose restrictions on the manner in which processes behave once execution commences. Specifically, the control flow among processes is neither controlled by TASK nor is it expressed in TASK programs.

In the next section, the style and flavor of the TASK language are communicated by presenting the essential language elements (see Appendix 1 for details). To motivate the choice of language elements, two design goals of the TASK language that differ from design goals elsewhere [62, 116] are discussed. The first goal is to vary the executable task forces constructed from a TASK program without the requirement of extensive changes in the program. The purpose is to facilitate experimentation. For example, in order to measure task force performance at different degrees of parallelism, task forces with different numbers of processes should be constructed. Such experimentation should not require revision of substantial parts of the associated TASK programs. The second goal of the design of TASK is not to burden programmers with the specification of construction detail. For example, programmers should not specify details such as the size of a task force component's runtime representation. Instead, construction information should either be derived automatically or set by default. The choice of language elements is also motivated by a design goal which is similar to those found elsewhere [62, 122, 116]: the incremental development of a task force. Specifically, the parts of each module not related to other modules in the task force should be independently specified, constructed, and tested.

The three design goals listed above are attained as follows. Changes in task force construction do not precipitate extensive changes in TASK programs because construction information is textually

separated from other information. For example, the specification of a process is textually separated from the instantiation of the process. As a result, the number of instantiated processes is easily changed. Similarly, construction detail that can be derived automatically or set by default is clearly separated from information that must be specified by programmers. Incremental task force development is possible because modules that are specified in the TASK language are individually linked, transferred to Cm*, and loaded.

2.2. A Task Force Example

Prior to describing the TASK language, an existing, experimental task force [29, 28] is presented. The associated TASK programs will be developed in the next sections. The experimental task force solves Laplace's partial differential equation with given boundary conditions. The PDE is solved by the method of finite differences. Specifically, the equation

$$\Delta^2 z / \Delta x^2 + \Delta^2 z / \Delta y^2 = 0$$

is solved for all points of an m by n rectangular grid. The solution is found iteratively, where in each iteration the new value of any grid point is set equal to the arithmetic average of the values of its four adjacent neighbors.

A parallel solution is attained by dividing the grid into a number of partitions. For each partition, a *server* process is created. Each server process has access to a set of objects that contain its code, storage for intermediate results, its partition of the grid, and those grid partitions that share boundaries with its partition. A server's code object contains replicated code. Since the PDE algorithm does not require synchronized grid accesses[1]. each server can run at its own speed. A *coordinator* process governs the task force. The coordinator communicates with server processes and with the user terminal. In addition, the coordinator initializes the grid array so that the PDE can be solved repeatedly. Communication with servers is performed by means of a communication object called

[1] In his thesis, Gerard Baudet shows that an appropriate PDE solution algorithm will converge to a solution even if the grid accesses of the server processes are not synchronized [7].

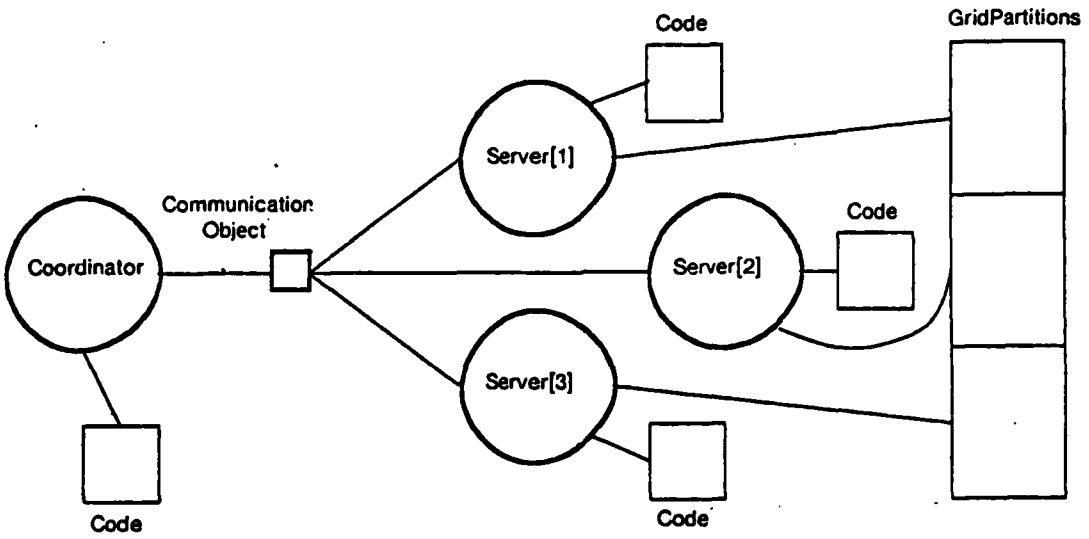


Figure 2-1: The PDE Task Force With Three Server Processes

"Commune" to which all processes share access.

Henceforth, the experimental task force presented above is called the PDE task force. To summarize, in the executable PDE task force, multiple, replicated server processes cooperate closely by iterating over the grid until a solution to the PDE is found. Data and code are distributed across processes by replication and partitioning. A sample executable PDE task force is shown in figure 2-1, in which one coordinator and three replicated server processes are displayed. The lines in the figure are drawn to indicate the accessibility of objects to processes.

2.3. The TASK Language

2.3.1. Basic Language Constructs

Recall that a task force is defined as a set of communicating processes and associated code and data objects, where processes cooperate to achieve a common objective [81, 80]. In TASK programs, processes and code or data are described as typed objects [163, 162, 78, 121, 43]. The type and name of each object and the attributes required for object construction, such as object size, are specified in a *Template*.

The presentation of templates and of other language constructs will involve syntax specifications in which three superscript symbols are used to denote different types of repetition:

- * means "zero or more repetitions of",
- + means "one or more repetitions of", and
- # means "either zero or one instance of".

When lists of items separated by some particular punctuation mark are to be denoted, the punctuation mark is indicated directly before the repetition character. The symbols { and } are meta-brackets and are used to group constructs in the BNF notation. However, meta-brackets are elided if parentheses or brackets required by the syntax already group the program constructs to which the repetition symbol applies. In the syntax descriptions, keywords are boldfaced, and ellipses indicate missing text. In the boldfaced program examples, keywords are underlined.

Templates. Templates are analogous to type declarations in languages like Pascal and Ada. Specifically, there are two kinds of templates in TASK: simple and complex templates. A *Simple Template* resembles a scalar type since it describes an object that does not have other objects as components. For each type of object, a different set of object attributes is specified. A *Complex Template* is similar to a record type because it specifies an object with components, such as a task force, a module, or a process. The syntax for templates is as follows:

⟨Template⟩ ::= {*⟨Complex Template⟩*|*⟨Simple Template⟩*}

⟨Simple Template⟩ ::=

<Simple Object Type> <Template Name> (<Actual Attributes[2]>)#

The type of a template determines the attributes that appear in it.

<Complex Template> ::= *<Task-Force Description>*
| ...
| *<Module Description>*
| *<Function Description>*

Function descriptions are prototype process descriptions (explained later in this section).

<Simple Object Type> ::= **Basic | Mailbox | ...**

The types of simple objects that are supported in TASK are just those object types that are supported by the STAROS operating system [149]. For example, the `Basic` objects (vectors of bytes) which contain code or data are the most frequently used STAROS objects. `Mailbox` objects are used in STAROS for sending and receiving messages. STAROS also directly supports the complex object types in TASK (for details, see Appendix 1 and [149]). Examples of simple and complex templates are presented in turn.

Simple Templates. As an example of a simple template, consider the specification of one grid partition in the PDE task force:

```
Basic GridPartition (Size=4K, Source=("Grid.Obj"))
```

This template has the name `GridPartition`. It describes an object of type `Basic` that contains up to 4K bytes of grid data. Presumably, the grid data with which object contents are initialized is found in the file `Grid.Obj`.

The *instantiation* of a template in a TASK program will result in the construction of the instantiated object by the loader. Instantiation of a template is analogous to the declaration of a variable of a certain abstract type in abstraction languages [164]. As with declarations of variables, template instantiations are separated from template specifications so that templates can be instantiated repeatedly. Consequently, the repeated construction of objects of the same kind is straightforward. Instantiations are performed by means of the `New` construct. In the case of simple templates, a `New`

---

[2] Keyword parameters are used in a list of actual attributes-see Section 2.3.2 for more detail.

construct is either directly followed by a template, or the New construct refers by name to a previously defined template:

```
New Mailbox (Size = 40)
New GridPartition
```

The use of the New construct will result in construction of a mailbox that is able to buffer up to 40 messages. Note that the size attribute is type specific. Mailbox size is measured in number of messages, and Basic object size is measured in number of bytes. The second use of New refers to the simple template defined earlier. The compiler will treat it as a textual substitution of the body of the template GridPartition. At load-time, a basic object of 4096 bytes will be created, and it will be initialized to contain up to 4096 bytes of data from the source file Grid.Obj.

Complex Templates. A *Complex Template* describes both the data contents and the components of an object, such as a task force, a module, or a process. Data contents are described in *Attributes* specified as part of the template's *Formal Parameters*, and components are described in a *Construction Description*. Each component can in turn be described by a *Complex Template*, so that arbitrary tree structures can be specified in this fashion. A template component is described by a *Comp*(onent) *Name* and an *Operation*. To exemplify the syntax of *Complex Templates*, the syntax of a *Module Description* is presented. The description of a sample module on page 23 is an example of a *Complex Template*.

```
⟨Module Description⟩ ::=
        Module ⟨Template Name⟩ (⟨Formal Parameters⟩)# is
            ⟨Construction Description⟩
            ...

⟨Construction Description⟩ ::= Construct (⟨Component⟩ ;*)

⟨Component⟩ ::= ⟨Comp Name⟩ : ⟨Operation⟩
⟨Operation⟩ ::= New {{⟨Object Type⟩ (⟨Actual Parameters⟩)#}
                    |⟨Template Name⟩ (⟨Actual Parameters⟩)#)}
              | Ref ⟨Object Name⟩ ....
              | Use ⟨Object Name⟩ ...
              ...
```

Comp(onent) *Names* are known only within the template in which they are defined, whereas *Template*

Names are known globally within each TASK program. Note that the ellipses indicate that additional operations and syntax detail have been elided; they are not needed for this presentation. *Formal* and *Actual Parameters* are explained later.

Operations provide alternatives in component instantiation. For example, the **New** construct states that a component object with a certain name be instantiated, where object instantiation consists of creating the object as well as initializing its contents. If the object is a complex object, then initialization includes the instantiation of its component objects. As an alternative to the instantiation of a **New** object, a named pointer to an already existing object is created by **Ref**erencing the name of an existing object. Another alternative is provided by the **Use** construct with which an identical copy of an existing object is created. If this construct is used, the costly initialization steps that are part of **New** need not be performed. For example, the data or code contained in each **New** object need not be transferred to Cm* via the EtherNet. Instead, the existing contents of the **Use**d object within Cm*'s primary memory can be accessed.

Modules. Modules contain functionally related task force components [130, 36, 164, 60, 78]. In addition, each module has a number of functions which are templates used for the instantiation of processes [60, 164, 130, 80]. The runtime representations of modules (supported by STAROS) restrict the objects that can be accessed by processes instantiated from their functions. Specifically, each process can access the components of its function and the components of its function's module. Therefore, modules act as firewalls during task force execution.

In the TASK language, a module is specified by a *Complex Template* describing the module object itself, the objects instantiated as module components, and *Function Descriptions*. Unlike functions in conventional programming languages, a *Function Description* is a template that describes a prospective process including any component objects. The syntax of a *Module Description* in conjunction with its *Function Descriptions* is as follows:

<Module Description> ::=
Module *<Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
<Function Description>+
...

<Function Description> ::=
Function *<Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
...

Consider the *Construction Description* of a *Complex Template* describing the server module. In the example below, mailbox and basic templates defined outside the module template are referred to by name within the module's *Construction Description*:

```
Mailbox Communicate (Size = 40)
Basic GridPartition (Size = 4K, Source = ("Grid.Obj"))

Module Server is
Construct (
                MyPartition: New GridPartition;
                Commune: New Communicate;
                Code: New Basic (Size = 4K, Source = ("Server.Obj"));
          )
...
```

When the server module is instantiated from this template, each component is instantiated in turn. Therefore, the module constructed by the loader will contain three objects: the code executed by server processes, one grid partition, and the object called **Commune**.

Processes. Processes are instantiated from function templates. Such instantiations are performed with a special *Operation* in which the function template is referred to by name:

<Operation> ::= ...
| **Process** *<Function Name>* (*<Actual Parameters>*)
...

Consider the sample instantiation of a single process for the function **DoServe** of the server module. In this example, the process is instantiated as a component of the server module. The function is defined a few lines below:

```
...
Module Server is
Construct (
        OneServer: Process Server.DoServe;
        ...
        )
        Function DoServe is
        Construct (
                Stack: New Basic (Size = 4K);
                ...
                )
...
```

A function is referred to by a pathname that consists of the module name followed by the *Function Name*. This permits functions in different modules to have the same name. *Function Names* (like other template names) are global so that a process can be instantiated anywhere within a task force using any function declared within the same TASK program[3]. Since process initialization consists of the instantiation of the components described by the *Function Description*, each server process will contain a **New Stack** object.

Since each process instantiated from a module's function can access the module's components, all processes instantiated from one module share access to the components of the module. As a result, a module object can be employed as a repository for the information common to its functions. For example, consider the refined function **DoServe** in which a component called **MyCode** is instantiated. Given this description of the function, each server process will contain a **New Stack** object and an exact copy of the server module's component **Code**:

[3]The current implementation of TASK further restricts process instantiation (see Appendix 1).

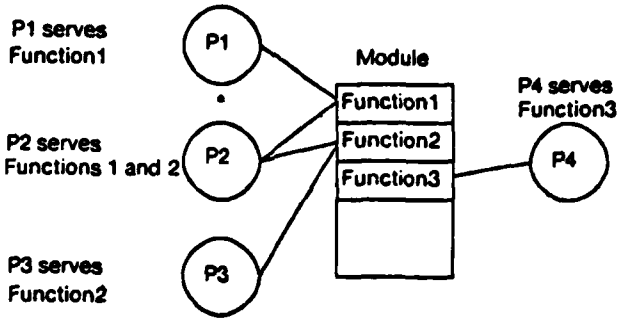


Figure 2-2: Processes Serving One, Multiple, or Overlapping Functions

```
...
Module Server is
Construct (
        OneServer: Process Server.DoServe;
        Code: New Basic (Size = 4K, Source = ("Server.Obj"));
        ...
        )
        Function DoServe is
        Construct (
                Stack: New Basic (Size = 4K);
                MyCode: Use Code;
                ...
                )
...
```

Similar ways of sharing within modular structures are also used in other languages. For example, in the MESA language [116], global information is specified in one *configuration file* shared by all processes in a *configuration*.

A function template is a flexible mechanism for process construction because multiple processes can be instantiated from one function. As a result, programmers need not substantially alter their TASK programs if the number of instantiated processes is changed. However, this mechanism can be too restrictive when performance concerns exist. Specifically, it is often desirable to instantiate a single process that fields the invocations of multiple functions in a module. For example, in a module

that implements the data abstraction **Stack**, it might be appropriate to instantiate a single process to serve the functions **Push** and **Pop**. To achieve such flexibility, we permit a function to be *aliased* to another function in the same module [149]. An invocation of an aliased function will automatically be converted to an invocation of the function's alias. In the example of the stack implementation, a function called **Both** could be the alias of the functions **Push** and **Pop**. Note that further augmentation of the function mechanism might be desirable. For example, aliasing is not sufficiently general to permit the instantiation of four processes from three functions, where three processes (P1, P2, and P4) each serve a single function, whereas a fourth process (P2) serves as a backup for two of the three processes. We graphically describe this situation in figure 2-2.

Formal Parameters. The access of processes instantiated from a module's functions to the components of the module is not the only means by which processes share objects. The other means is to declare *Formal Parameters* to module or function templates. A *Complex Template* with *Formal Parameters* is analogous to a parameterized type definition in an abstraction language. To provide an example of a parameterized template, the mailbox called **Commune** is declared a parameter to the module template **Server**. The server process instantiated from the function **DoServe** is presumably programmed to cycle, while processing work requests sent to it via the mailbox **Commune**. Each work request instructs the server process to continue processing the data in **MyPartition** until either a given time limit is exceeded or the solution to the PDE has been found (see example 2-1).

```
...
Module Server (Commune: Mailbox) is
Construct (
        OneServer: Process Server.DoServe;
        MyPartition: New GridPartition;
        Code: New Basic (Size = 4K, Source = ("Server.Obj"));
        )
        Function DoServe is
        Construct (
                Stack: New Basic (Size = 4K);
                MyCode: Use Code;
                MyCommune: Ref Commune;
                )
...
```

Example 2-1: The Parameterized Server Module

In the example 2-1, each server process will contain a **New Stack** object and an exact copy of the server module's component **Code**. However, the mailbox called **Commune** is only **Referred** to within the function template so that all server processes will share access to it. Since **Commune** is a parameter of the module[4], it is presumably instantiated elsewhere. In later specifications of the PDE task force, the mailbox **Commune** will be used for communication between the server processes and the coordinator process. Toward that end, it will also be declared a parameter to the coordinator module.

Task Forces. Recall that a TASK program describes a task force as a collection of modules. Module templates are instantiated as components of a *task force template* that represents the entire task force. The form of a task force template is similar to that of any *Complex Template*, except that it cannot have any parameters:

TaskForce *<Complex Template Name>* **is**
<Construction Description>
. . .

In order to exemplify a task force template, we use the server module template defined in example 2-1 and we assume that a similar template has been defined for the coordinator module. The latter template exports the function **DoCoordinate**, and the coordinator process is presumably instantiated as a component of the coordinator module. Both modules are instantiated within the task force object. The communication object **CommObj** is specified as an *Actual Parameter* to both modules so that it is accessible to both the coordinator process and the server processes. Presumably, the process executing **DoCoordinate** will hand work requests to server processes via the **CommObject**. The template for this task force, called the **PDE** template, can be found in example 2-2.

```
TaskForce PDE is
Construct (
        CommObject:             New Mailbox (Size = 40);
        CoordinatorModule:      New Coordinator (Commune=CommObject);
        ServerModule:           New Server (Commune = CommObject);
        )
```

[4]Formal parameters to the module template can be referenced from inside each of its function templates.

```
...

Module Server (Commune: Mailbox) is
Construct (
        OneServer:      Process Server.DoServe;
        Code:           New Basic (Source=..., Size=...);
        MyPartition:    New Basic (...);
        )
        Function DoServe is
        Construct (
                ...
                MyCommune: Ref Commune;
                ...
                )
...

Module Coordinator (Commune: Mailbox) is
Construct (
        OneCoordinator: Process Coordinator.DoCoordinate;
        Code: ...
        )
        Function DoCoordinate is
        Construct (
                ...
                MyCommune: Ref Commune;
                ...
                )
...
```

Example 2-2: The PDE Task Force

When a task force is instantiated from the template PDE, a task force object is created and all of its components are instantiated, including the two modules. The instantiation of each module causes the instantiation of its components, thereby causing the instantiation of a process. Only slight alterations of the templates in example 2-2 are required to instantiate the grid partition as a component of the task force object. In that case, the grid partition can be passed as a parameter to both the server and the coordinator module so that both modules share access to the partition.

To conclude, we note the differences between the TASK specification in example 2-2 and the executable task force constructed from it (see figure 2-1). In the TASK program, the task force is described in terms of the instantiated modules, whereas the executable task force is described in terms of the instantiated processes which merely make use of module objects. Fundamentally, these differences arise because TASK programs focus on task force structure and construction, whereas

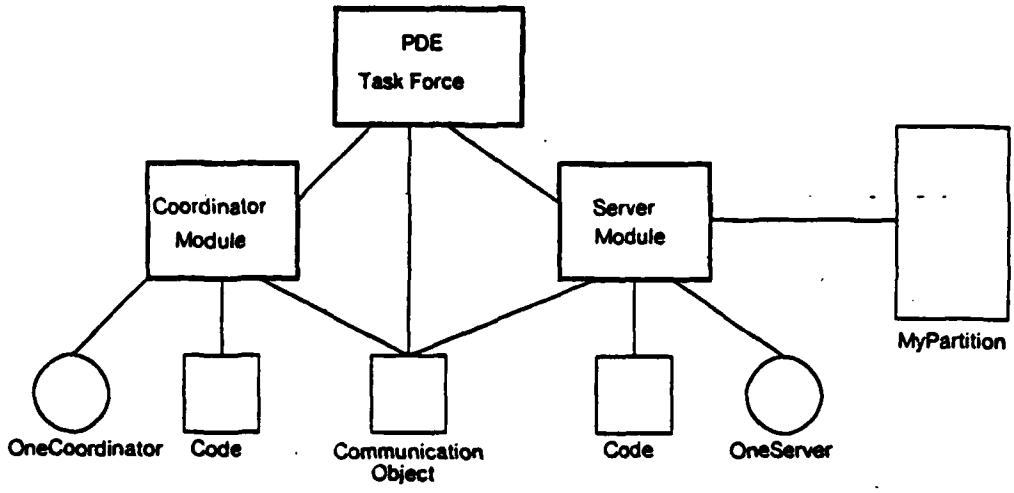


Figure 2-3: The PDE Task Force's Coordinator and Server Modules

the descriptions of executable task forces are dominated by the activities of their processes. Task force structure in terms of *component of* relationships between objects must be shown since the construction of any object will precipitate the construction of its components. *Component of* relations also determine the scope of *Comp*(onent) *Names*. In addition, construction detail, such as object sizes, must be contained in TASK programs. In executing task forces, object names, their scopes, and therefore, *component of* relationships need not be known. Furthermore, construction detail need not be retained. Instead, the accessibility of objects to processes must be known (as shown in figure 2-1). In TASK, each process can access its components and the components and parameters of its function's module.

The differences between an executable task force and a TASK program are emphasized by a comparison of figure 2-1 with a figure in which the TASK text is displayed graphically (see figure 2-3). In the latter figure, the two module objects and the task force object are displayed. Each module has three components: a process object, a code object, and the communication object. The server

module contains an additional object: the grid partition. Note that the communication object is a component of both modules as well as of the task force.

2.3.2. Attributes and Iteration

One of the design goals of TASK is not to burden programmers with the specification of construction detail. To attain this goal, construction detail is either set by default or derived automatically whenever possible. To distinguish information that must be specified by programmers from construction detail, both kinds of information are textually separated within TASK programs.

Attributes. Construction detail is specified in object *Attributes*. Consider the template called **Communicate**. The type and name of the template must be specified by the programmer, whereas the attribute values such as object size can be set by default. Consequently, the minimal specification of **Communicate** is one in which no attributes are contained:

```
Mailbox Communicate
```

As a result, the mailbox with the following default attribute values is constructed:

```
Mailbox Communicate (Size = 32, MsgType = Data)
```

The **Size** of the mailbox specifies the number of messages that can be stored (32). **MsgType** distinguishes mailboxes that contain data messages from mailboxes that contain structured messages [149, 134]. The default value **Data** is chosen since mailboxes of this type are most frequently used in task forces executing under the STAROS operating system.

In TASK, most attributes of objects are set by default (see Appendix 3 for a complete list of attributes and defaults). However, the object attributes that are dependent on the logic of the algorithmic code must either be specified by programmers or derived by the linker. To clarify, we provide examples.

Consider the attribute **Size** of a **Basic** object that contains a specific data structure. If the data structure is allocated at compile-time, the TASK linker can determine the number of bytes it contains, whereby the loader can be furnished with the size of the basic object to be constructed [123]. In this

case, a programmer need not specify the value of the basic object's attribute **Size**. However, if the data structure is constructed at runtime, then the compiler, linker, or loader cannot determine the basic object's size. Therefore, the object's attribute **Size** must either be specified by the programmer or the default supplied by TASK must be used. Note that the size of objects containing code can always be determined by the linker.

Next, consider the attribute **Source** naming files that describe the contents of objects. The values of this attribute cannot be derived automatically or set by default because TASK programs do not contain information concerning the logic of the algorithmic code. For example, TASK programs do not contain information concerning the values of compile-time constants used by the algorithmic code. If such constants are contained in a basic object, the programmer must specify the basic object's attribute **Source** in which a file containing the constants' values is named.

We note that illegal attribute values, such as negative sizes, are detected by the TASK compiler and corrected by insertion of default values. Furthermore, if parameters remain unspecified, the compiler issues warnings and inserts null parameters so that construction can proceed.

Another design goal of TASK is the variation of task force construction without requiring extensive changes in TASK programs. Small construction changes are possible by variation of object attributes. However, changes such as the variation of the number of instantiated processes involve the addition or deletion of component instantiations in the TASK program. We call such changes *structural* task force changes. A straightforward structural change in a TASK program is the variation of a replicated or partitioned component. In a *Replicated Component*, a variable or fixed number of identical objects are instantiated from one template. Each object has a unique name. For example, it can be stated that several processes are to be instantiated from a function template. Their number need not be determined until the task force is loaded [84]. In a *Partitioned Component*, a variable or fixed number of objects are instantiated from one template. Each object has a unique name and contains an equal fraction of the data or code stored in the template's source file. For example, partitioning the grid

data separates the grid into several grid partitions of fixed size (one of which is shown in the PDE program on page 28). The number of partitions depends upon the amount of grid data in `Grid.Obj`. This number need not be determined until load-time.

Iteration. In TASK programs, *Partitioned* or *Replicated Components* are stated with an iteration construct. In the following example, the grid is divided into n partitions of size "4096" bytes each. The variable n is a formal parameter of the template in which the grid partitions are instantiated. Since the value of this parameter must be determined when the template is instantiated, the following partitioning statement resembles a variable macro expansion:

```
(i=0..n) Grid[i]: New Basic (Size=4096)
```

This statement is expanded to:

```
Grid[0]: New Basic ...
Grid[1]: ...
...
Grid[n]: ...
```

We refer to appendix 1 for a detailed definition of TASK's iteration construct.

In the current implementation of TASK, only object replication and partitioning variables can remain unbound until load-time. All other attributes, variables, and parameters are resolved during program compilation.

2.3.3. The Interface Between TASK and BLISS

Bliss programs are not visible in a TASK program with the exception of the names of files in which they are contained. However, since Bliss programs manipulate the objects named within a TASK program, the two kinds of programs must share names for the same objects. Since integers are more space-efficient than strings, the names shared are integer names which are assigned to the *Comp*(onent) *Names* in the TASK program.

The implementation of name sharing is constrained by the requirement that TASK-generated task forces must be compatible with task forces written in Bliss-11 prior to the implementation of TASK.

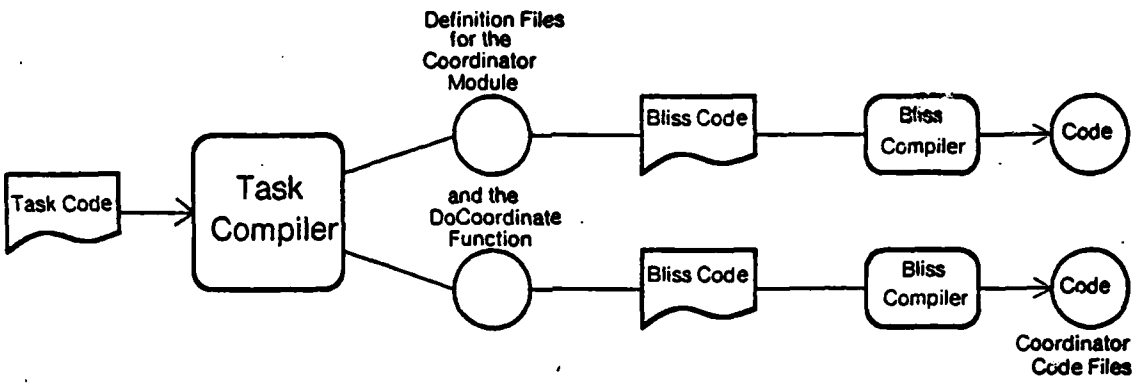


Figure 2-4: Generating and Using the Definition Files of the Coordinator Module

Consequently, the interface we have implemented does not require changes in Bliss. Specifically, the TASK compiler generates *definition files* that are compiled with Bliss programs. These definition files contain Bliss macro definitions in which integer names are bound to *Comp Names*. As a result, TASK and Bliss programs must be recompiled whenever the contents of definition files change. However, small changes in either programs are not likely to affect the mapping of *Comp Names* to integers so that typically TASK and Bliss program alterations can proceed independently of each other.

Recall that another goal of the design of TASK is the construction of a task force module by module. Accordingly, it must be possible to compile a task force incrementally. Toward this end, TASK generates multiple definition files from one TASK program: one for the task force object, one per module and one per module function. For example, two definition files are generated for the coordinator module: one listing the module components and the other listing the components of the module's function (see figure 2-4). These files act as input to separately compiled Bliss programs which are later linked to become the code executed by the coordinator process.

2.4. Task Force Blueprints

When programmers write TASK programs, they typically first design the logical task force structure and write the associated code. Then, construction-specific information is varied to perform a variety of experiments, while the fundamental structure and the code of the task force remain fixed. For example, programmers may run a task force with different numbers of processes in order to measure performance at different degrees of parallelism. To characterize the variety of executable task forces that can be constructed by straightforward changes of a single TASK program, we introduce program blueprints [83]. As with engineering blueprints, a *blueprint* of a TASK program describes some of the instantiated objects and specific relationships between those objects [114, 169]. Just as in engineering, different blueprints of the same TASK program are used for different purposes. Specifically, three blueprints of a single TASK program are defined. The *logical blueprint* is a specification of task force structure that contains a description of each object instantiated within the TASK program. This blueprint contains the information that typically remains unchanged across experiments with the executable task force. Specifically, parameter and attribute values are not contained in the logical blueprint. The information that varies across experiments is captured in the *execution blueprint*, which is defined as a logical blueprint in which all parameters and attributes are bound. A third task force blueprint, the *proximity blueprint*, will be defined in Chapter 3.

2.4.1. Logical and Execution Blueprints

Logical and execution blueprints are described in turn. As previously stated, the *logical blueprint* contains a description of each object in the executable task force. An object description consists of the object type and string name. Object type will determine the permissible object attributes. The TASK language is used to state the information specified in a logical task force blueprint. Consider the following TASK text that corresponds to the description of the grid partition in the logical blueprint of the PDE program (see example 2-2):

```
GridPartition: New Basic
```

This sample entry in the logical blueprint consists of the grid object's string name, the operation

stating that a new object will be instantiated, and the object type. Note that the string or integer attribute values are not specified.

A logical blueprint reflects task force structure by recording the *component of* relationships between instantiated objects. Object "b" is the *component of* object "a" if "b" is instantiated as a component of the complex template describing "a". For example, if the grid partition above is instantiated as a component of the server module, then the logical blueprint records a *component of* relation between the server module instantiation and the grid partition instantiation. Iterated components are described by a single entity in the logical blueprint. For example, the partitioned grid is considered a single *component of* the server module. Processes are described like any other object. They are the *components of* the objects in which they are instantiated, and their own components are described by the function templates employed for process instantiation.

If a template has *Formal Parameters*, the *component of* relations hold for any object bound to these parameters. However, *Actual Parameters* are not specified in the logical blueprint. Consequently, while the object bound to the formal parameter **Commune** will be considered a *component of* the server module, this object is not known in the logical blueprint. Note that **Commune** is an indirect parameter of the function **DoServe** (see **MyCommune** in example 2-2). Therefore, the object bound to **Commune** will also be considered a *component of* each server process.

While *component of* relations do not hold for the non-object *Formal Parameters* of a template, the names of these parameters (but not their values) are contained in the template's description in the logical blueprint. Non-object parameters must be known in the logical blueprint because such parameters can affect the structure of an object instantiated from a template. Recall that a structural change of a TASK program is one in which the number of instantiated components is varied. Consequently, a sample parameter affecting structure is one that varies the bound of a vector of instantiations. Conversely, a parameter that does not affect structure is one that varies an attribute value. Consider the following TASK text:

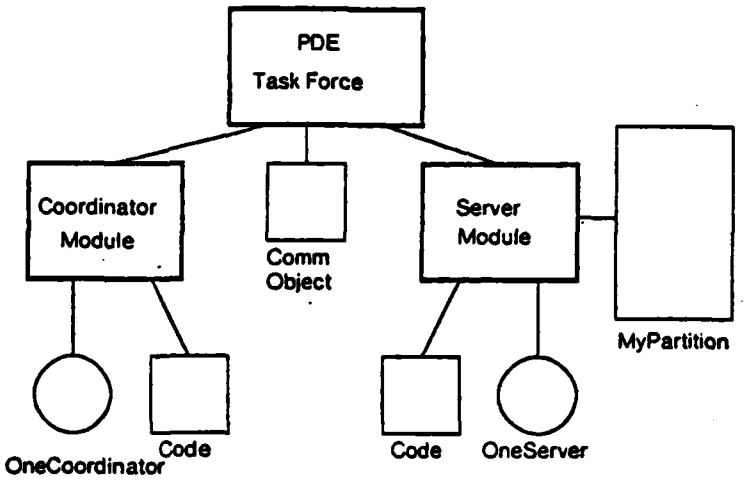


Figure 2-5: A Logical Blueprint of the PDE Task Force--Process Components are Elided

```
ServerModule: New Module Server (n:Integer, ObjSize:Integer) is
                . . .
                (i=0..n) Grid[i]: New Basic (Size=ObjSize, Source= ...)
                . . .
```

In this program fragment, the template parameter n is the upper bound of the vector of instantiations `Grid[i]`, which is a single entity in the logical blueprint. The parameter `ObjSize` is employed to vary the size of each entry in the vector of instantiations. It does not affect the structure of the server module.

We summarize the discussion of the logical blueprint by referring to the excerpts of the TASK program 2-2 displayed in figure 2-5. Since the attribute and parameter values found in 2-2 are excluded and since the parameter `CommObject` to the server and coordinator modules is not resolved, this TASK text represents a logical blueprint of the PDE task force. However, for simplicity, the function specifications and therefore, the components of the processes in the PDE task force have been elided. A graphical representation of the TASK text is shown next to it. The lines in this figure (i.e. figure 2-5) represent the *component of* relations between object specifications.

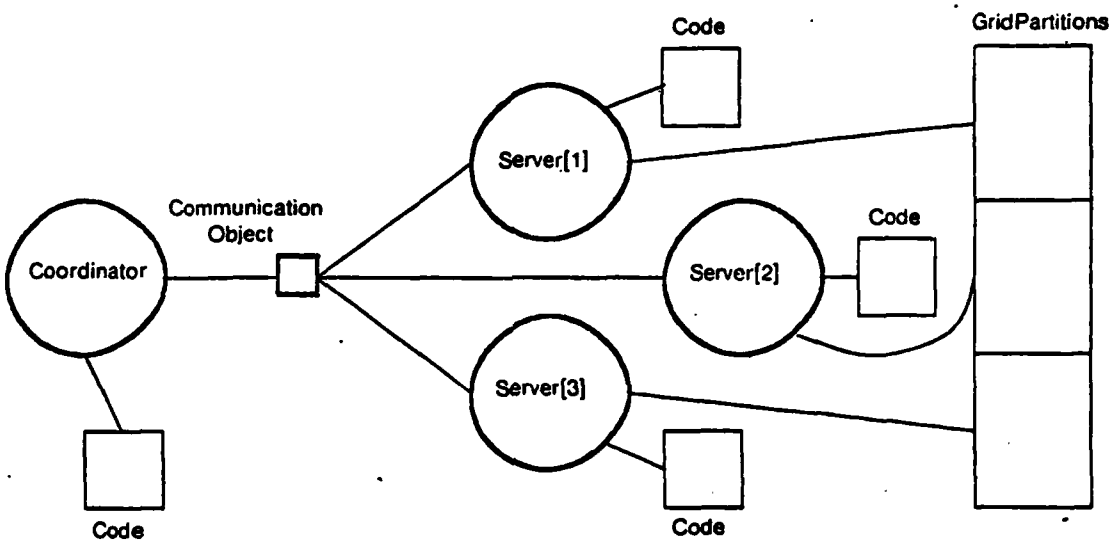


Figure 2-6: The PDE Task Force With Three Server Processes

Next, we discuss the execution blueprint, which is defined as a logical blueprint in which all attributes and parameters are bound to values. Therefore, the execution blueprint contains all information required for task force construction. Specifically, each execution blueprint includes the descriptions of the modules, the processes, the replicated and partitioned objects, and the object parameters in the TASK program. Replicated and partitioned objects are represented in an expanded form. For example, the vector of instantiations `Grid[1]` is represented as "n" separate entities, where the value of "n" is known. Again, *component of* relations are specified between object instantiations.

As an example of an execution blueprint, consider the executable PDE task force that consists of four processes: one coordinator and three replicated server processes. The coordinator process is a component of the coordinator module, and the replicated server processes are components of the server module. The communication object is a component of each process because it is both an

actual parameter of the respective module instantiations and is Referred to from within the respective function descriptions. In figure 2-6 the processes in the execution blueprint are displayed, whereas the specifications of the task force and module objects as well as the detailed construction information are omitted. The lines represent *component of* relations.

Having defined logical and execution blueprints, the degree to which TASK programs are affected due to experiments with the executable task force can be stated. For example, several experiments with the PDE task force will entail the instantiation of different numbers of server processes and grid partitions, while other experiments will change the association of grid partitions with servers [29] (see Chapter 3). The logical blueprint and the fundamental task force structure remain unchanged in either experiment. First, if the server processes are instantiated as a *Replicated Component* and if the grid is a *Partitioned Component*, then the number of server processes and grid partitions is varied by adjusting the values of replication and partitioning parameters. Since parameter values are not contained in the logical blueprint, this blueprint remains unchanged. Second, if each server process can access the entire grid, then logical blueprint changes are unnecessary when altering the association of grid partitions with server processes. Instead, server code can determine the proper association.

Although the sample experiments leave task force structure unchanged, construction of the executable task force is affected. Specifically, each experiment's execution blueprint contains different values of replication and partitioning parameters. However, the association of grid partitions with server processes is determined by server code, and therefore, is not visible in execution blueprints.

The logical and execution blueprints can even remain unchanged when the reference of a process to a specific *Function Description* is changed. For example, whether the process called `OneServer` is instantiated from the functions `DoCoordinate` or `DoServe` is not visible in the case in which both function templates contain the same components. However, the *use* of a function to instantiate a process [131] affects the manner in which the task force is constructed. For example, consider the

original design of TASK in which any process could be instantiated from any function defined in the same TASK program. This design exhibited complications concerning incremental task force construction. Specifically, if two modules *used* each other's functions to instantiate processes, then neither module could be constructed before the other or each module had to be re-initialized once the other had been constructed. Furthermore, if a function invoked itself, then initialization would cycle.

Two approaches can be taken to avoid the re-initialization of modules and cyclic dependencies between processes. In one approach, the logical and execution blueprints are augmented to contain *Uses* relationships so that cyclic dependencies can be detected [63]. In addition, the loader is extended to perform repeated initializations. In the other approach, the scope of function names is reduced such that a *Function Name* is known only within the module in which it is defined. As a result, a process is always instantiated within the module that defines its function. Note that users must build "self-sufficient" modules containing all information needed by their processes. In addition, processes in different modules share access only to their module's parameter objects. For simplicity, the second approach is chosen in TASK.

We note that TASK enforces comparable, but less rigid, restrictions concerning the *Uses* relationships between any object and any template in a TASK program.

2.4.2. Representing Blueprints within the TASK Compiler

The current implementation of the TASK compiler requires the recompilation of TASK programs whenever they are changed. However, we can show that the TASK compiler could be readily extended to permit certain changes in task force construction without TASK program recompilation. The simplicity of this extension is due to the manner in which TASK program blueprints are represented within the TASK compiler. The representations of logical and execution blueprints are discussed in turn.

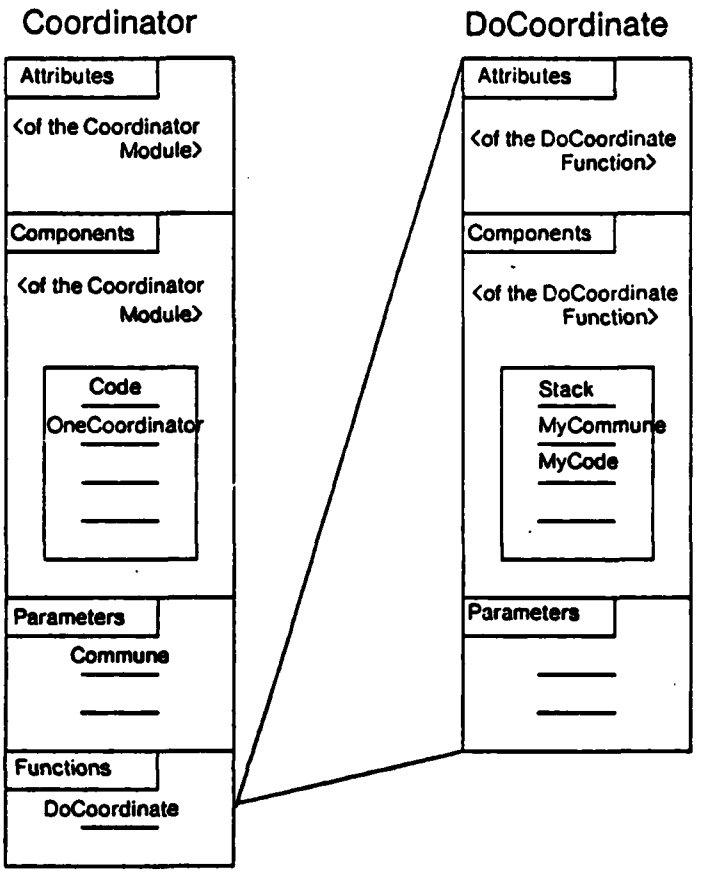


Figure 2-7: The Templates of the Coordinator Module and its Function

The logical blueprint is represented as a shallow forest of templates. Function templates exist within module templates, whereas all other templates are at the top level. Each template is specified once, regardless of the number of objects instantiated from it. Within a template, a single entry is made for each individually instantiated object and for each vector of instantiations. The coordinator module and its function template are graphically displayed in figure 2-7. The components and the formal parameters of the coordinator module are displayed, while attribute values are elided. For simplicity, the template for the `DoCoordinate` function is shown separately from the module

template. Note the parameter Commune in the coordinator module.

The representation of the logical blueprint chosen is efficient in space because the total number of templates in any given TASK program is typically small. In contrast, the number of object instances within a single task force can be large. If templates were physically duplicated when used for object instantiations, the compiler's available storage would soon be exhausted. To avoid template duplication, a *link* [154] is specified from each instantiated component to its template. For parameterized templates, actual parameters are maintained with each instantiation. Actual parameters are matched with formal parameters when the *linked* object is instantiated.

The information in the execution blueprint is complete when the TASK compiler's semantic processing phases have bound all attributes and parameters in the template forest. Task force construction could proceed based on the completed template forest. However, to explicate the task force constructed from the given template forest, the TASK compiler maintains another representation of the execution blueprint; it is called a *creation tree*. The top node of this tree represents the task force object itself. Descendant nodes represent task force components. Each complex component is a node in the tree that gives rise to another subtree. Simple components are leaves of the creation tree. Each vector of instantiations is expanded resulting in the number of nodes or leaves indicated by the upper bound. A pointer within each node or leaf refers to the detailed object description in the template forest. In figure 2-8, the part of the tree constructed for the coordinator module is displayed. The lines in the figure illustrate tree arcs encoding *component of* relations.

Several benefits are derived from the creation tree[5]. Most importantly, the creation tree is an explicit representation of the task force to be constructed. In this representation, a unique object in the executable task force corresponds to each node or leaf in the creation tree. This is not the case in the template forest, in which a single template entry can represent a vector of instantiations, and in which a single template can be used for several object instantiations. Due to the one to one cor-

[5]Creation trees in TASK can be compared to syntax trees in syntax-oriented language editors [113].

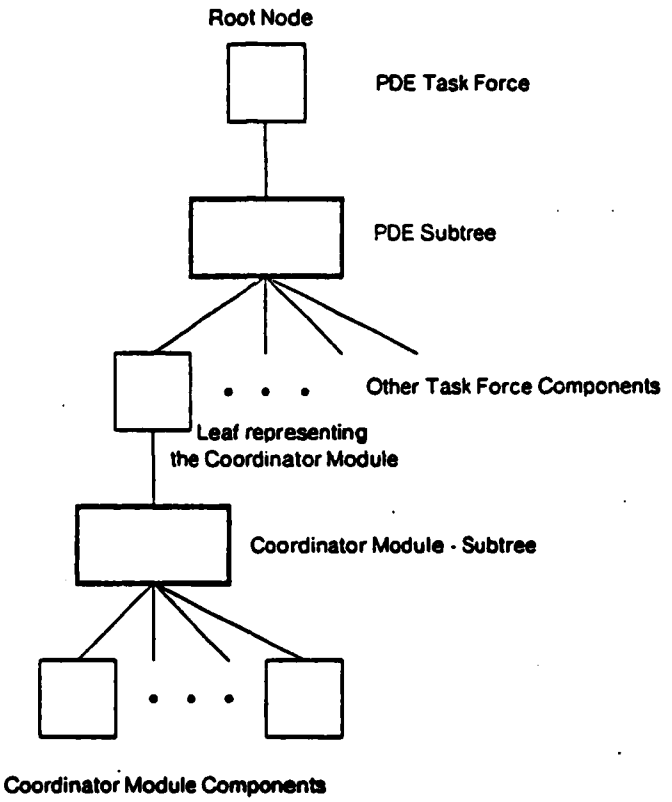


Figure 2-8: The Partial Creation Tree of the Coordinator Module in the PDE Task Force

respondence between creation tree entries and task force objects, a comparison of the task force to be built with the task force actually constructed is straightforward. Such comparisons are useful for task force tailoring, monitoring, and debugging (see Chapters 4 and 5). The creation tree is also useful concerning the generation of target code. For example, the generation of linking and loading instructions each involve a straightforward tree traversal.

The creation tree and the template forest are data structures in which certain changes in task force construction are straightforward to represent. Therefore, the associated TASK program need not be recompiled. Specifically, structural changes consisting of variations in component replication and partitioning are represented by updates to the replication or partitioning values in the template forest,

whereupon the creation tree is rebuilt. Changes in construction detail, such as object sizes or source attribute values, involve updates to individual entities in the template forest, while the creation tree remains unchanged.

2.5. Task Force Construction

Recall that task force construction consists of compiling Bliss and TASK programs, and of linking and loading (see figure 1-1). To automate construction, the TASK compiler generates instructions that are stored in linker or loader command files and are carried out by the linker or the loader. In support of incremental construction, individual command files are generated for each module in the TASK program, and for the task force itself. To clarify, we illustrate the cooperation among the TASK compiler, Bliss compiler, linker, and loader in figure 2-9. Specifically, we display linker and loader command files as well as the definition files generated for Bliss programs. Dashed lines indicate the control of linker and loader by means of command files, whereas solid lines indicate the flow of information through the linker and loader.

In the remainder of this section, the instructions generated by the TASK compiler and the processing of these instructions by the linker and loader are discussed. In addition, the integration of debuggers and monitors into TASK is noted.

2.5.1. The Linker

The algorithmic code of each module in a TASK program consists of multiple, separately compiled Bliss programs stored in multiple files. However, these programs contain procedures that refer to each other and to common variables. Therefore, they must be linked before they can be loaded.

The TASK linker is an adaptation of the linker used in the Cmmp system [30]. In addition to linking Bliss programs, the linker collects the source files and the loader command files of each module into a single file, called a *page file*. A page file is formatted such that code, data, and commands are contained in a sequence of numbered pages of varying size. As a result, rather than accessing

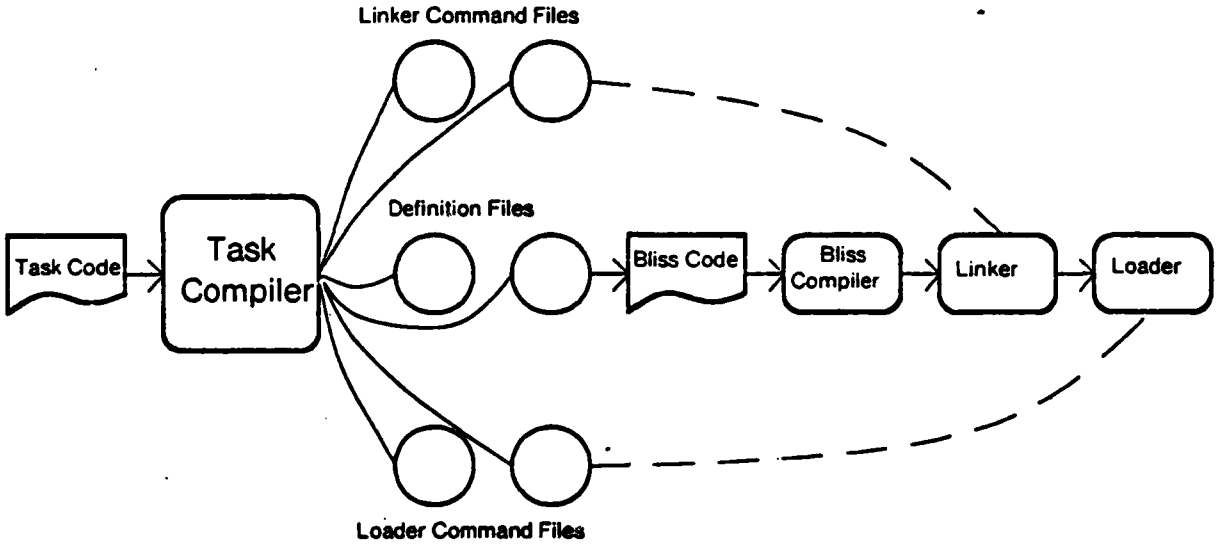


Figure 2-9: The Relationships among TASK's Compilers, the Linker, and the Loader

multiple files during the construction of a module, the loader need only refer to particular pages in a single page file. Since the linker is described elsewhere [30, 149], further detail concerning its operation will not be provided.

2.5.2. The Loader

In TASK, loading is automated and transparent to programmers. Consequently, programmers are relieved of this tedious, error-prone job, and task forces of any size can be constructed with comparable ease. However, loading cannot be automated unless resource allocation is performed automatically. Specifically, two resource allocation decisions have to be made prior to loading:

- *assignment*--on which processor to execute a process;
- *placement*--where in distributed memory to represent object contents.

Whereas other loaders make and execute the allocation decisions above[6], the TASK loader merely executes allocation decisions which are made prior to loading. The automation of resource allocation is discussed in Chapters 3, 4, and 5. In the remainder of this section, we describe the manner in which the loader executes allocation decisions while a task force is constructed.

Although resource allocation is decided prior to loading, the TASK loader remains a complex subsystem for the following reasons. First, the entire tree of components of a task force (see the creation tree in Section 2.4.2) must be constructed. Within this tree, components may **Refer** to or **Use** other components by name, and the branches of this tree that represent modules must be constructed separately from each other. Furthermore, recall that objects which are bound to parameters can be shared within modules or across module boundaries. Since formal parameters, **Ref**erences to, and **Use**s of other objects cannot be resolved before the corresponding objects have been constructed, either construction has to proceed in the appropriate order or objects must be initialized more than once. The latter solution is chosen to simplify the code generation phase of the TASK compiler.

Two additional sources of complexity in the loader are its interactions with the distributed operating system and with the executing task force [122]. Specifically, the loader must make use of the complex command formats and request protocols of the operating system which governs the substantial number of resources of Cm* [123, 138]. In addition, although the intended use of the loader is strictly static (i.e. permitting a task force to be constructed once and then executed), the loader also plays a dynamic role in task force construction. Specifically, the TASK loader will construct each process in the task force either statically or dynamically (see the next section).

In the remainder of this section, we will explain the contents of loader command files and the manner in which they are processed. In addition, the data structures maintained by the loader are

---

[6] At ISI, a "downloader" was developed to assist programmers in making detailed resource allocation decisions concerning a small number of graphics processors.

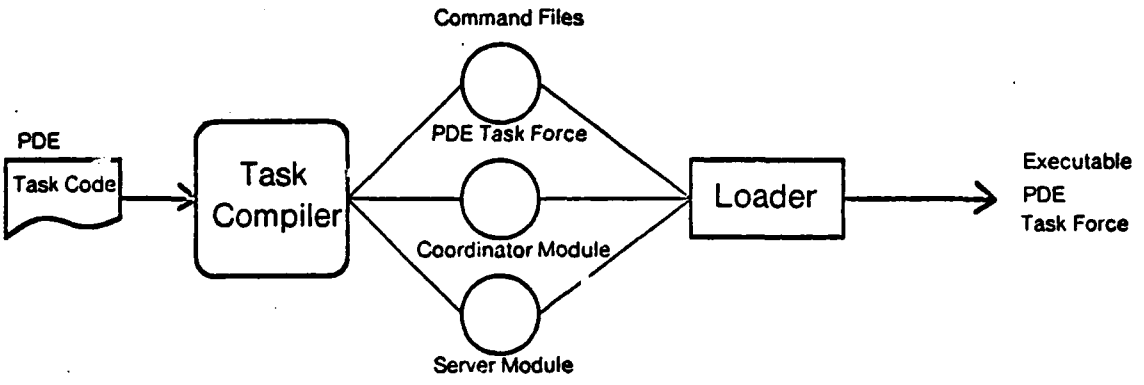


Figure 2-10: Using Command Files For Loading

presented. As examples, excerpts of the loader command files for the PDE task force will be shown.

Command Files and Command Templates. The loader command files of the PDE task force and the flow of control between the TASK compiler and the loader are illustrated in figure 2-10 (for simplicity, the linker is elided). Note that three command files are generated: one per module and one for the task force itself. The contents of these command files are analogous to the contents of TASK programs, in which templates are used to specify the information concerning each object in the task force.Specifically, a loader command file is formatted as a sequence of templates called *command templates*, where each template contains the information required for the construction of a single object:

- a specific command to differentiate among the instantiation of **New** objects and the **Reference** or **Use** of previously instantiated objects;
- a specification of the object stating the attribute values expected by the operating system's resource managers (e.g. object sizes) and the attribute values required for object initialization (e.g. the source of object contents);

We note that the attribute values within a command template can be declared parameters of the command template. Such parameters are resolved by the loader prior to their use.

Consider the creation and initialization of an object instantiated from a simple template of type **Basic**. A command template containing the information displayed below is generated (PDP-11 assembly language is used to encode a command template. Comments are preceded by "--". Precise definitions of each entry in loader command templates can be found in Appendix 4.):

```
-- A sample command template that instructs the loader to
-- CREATE an object called DATA of type BASIC as a component of
-- some template, in the template's component slot 1:

        -- The first word contains the specific command and
        -- a pointer to the object attributes

        .WORD 0,C$$$2 -- 0: a creation command, C$$$2: a pointer
        .WORD 1,0     -- the component slot (1) into which the
           -- pointer to the object is placed
        .BYTE 0,0,0,133 -- an "end of command" marker denoting the
           -- absence of actual parameters

-- The attributes of DATA:
C$$$2:
        .WORD 4096,0 -- an object of size "4096" bytes
        .WORD 0,0    -- the object has no component pointers
        .WORD 0,0    -- an entry expected by the memory allocator
        .BYTE 0,0,255,128 -- the object can be placed anywhere in
           --the distributed memory of Cm*
        .WORD 0,0,   -- the object is of type "0", the memory
           -- allocator's encoding for BASIC
        .WORD 1,0    -- object contents must be initialized
        .WORD 0      -- an "end marker"
```

When the loader interprets this command template[7], it detects the **Create** command (**0**). Prior to submitting a request for space to the operating system's memory allocator, the loader accesses the attributes (**C$$$2**) and the actual parameters (if any) in order to determine the particular memory unit in which to allocate space and the amount of space to be allocated. Subsequent to memory allocation, the loader initializes the object to contain the data within the object's source file. If such a file exists, its contents are found in a page of the page file currently used. In the command template shown, this is the next page, whereby a specific page number is not given. The object is considered fully constructed upon the completion of initialization.

---

[7] We note that these loader actions are not significantly different from the actions taken by user programs that explicitly instantiate objects.

Except for catastrophical failures, users are not involved in the process of object creation and initialization. To minimize the necessity of user interaction, the loader attempts to reduce the frequency of failures by detecting the loss of object creation requests made to the operating system. It uses a time-out scheme and automatically resubmits lost requests.

To illustrate the manner in which the loader constructs objects instantiated from complex templates, consider the construction of a module. As previously stated, a module is represented at runtime as an object that contains a vector of function descriptions and a list of pointers to module components [78]. The runtime representation of a module is constructed in two steps: first, the module object is created and then its data part and component pointers are initialized. Component pointers are initialized by instantiation of the module's component objects. Accordingly, the module's command template contains component instantiation commands in addition to the creation command for the module object itself. Excerpts of the command template for the server module in the PDE task force are presented in the following text:

```
-- A command template that instructs the loader to
-- CREATE the object called  SERVER of type MODULE
-- in component slot 1 of the task force object:

          .WORD 0,SERVER          -- 0: create the object at SERVER
          .WORD 1,0    -- the component slot in the task force (1)
          .BYTE 0,0,0,133 -- an "end of command" marker denoting the
                     -- absence of actual parameters


-- The attributes of SERVER:
SERVER:
          .WORD 256,0  -- the module's data part size is 256 bytes
          .WORD 32,0   -- it can contain up to 32 component pointers
          .WORD 0,0    -- an entry expected by the memory allocator
          .BYTE 0,0,255,128 -- the module can be placed anywhere
          .WORD 5,0    -- an object of type MODULE, "5"
          .WORD 1,0    -- the module's data part must be initialized
          .WORD SERVER$            -- the commands for constructing
             -- the module's components
```

```
-- The beginning of the command list:
SERVER$:
        . . .
        -- A sample command:
        -- Create DATA of type BASIC in module component slot 1
        . . .
-- The end of the command list, an "end marker":
.WORD 3
```

Given this command template, the module object will be created as a component of the task force. Initialization of module components is described by a sequence of initialization commands, such as the previously explained construction command for the object called **Data**.

The Command Language. The loader command language resembles job control languages in operating systems [166, 145]. Typically, object construction involves the invocation of operating system resource managers and is performed in the order in which construction commands appear in command files. However, if several objects of the same kind are to be constructed, a sequence of creation commands can be executed repeatedly. Furthermore, if an object is to be initialized once and re-initialized later, the initialization commands of the object's command template can be executed once and re-executed later, whereupon only those initialization commands not yet successfully completed are executed. In addition, object copies or pointers to other objects can be fabricated. (For additional detail concerning the loader's command language, consult Appendix 4).

The resemblance of the loader command language to job control languages suggests that it is straightforward to extend the TASK loader so that task force components can be loaded before and during execution. This extension is pursued further in the next section.

Repeated Initialization. Since actual parameters are not necessarily constructed at object initialization, the loader must permit the repeated initialization of an object. A possible use of the repeated initialization of an object occurs during task force construction. Specifically, a task force could be constructed and then repeatedly initialized and executed, either to debug or to repeat performance measurements. Since the command templates used by the TASK loader are not retained until task force execution, the TASK system cannot assist users in the dynamic initialization of objects.

However, the data structures required to implement this dynamic initialization are similar to a data structure already maintained by the TASK loader, called the creation stack.

The Creation Stack. The *creation stack* is used to initialize an object at times other than immediately after object creation. It contains pointers to command templates and to the associated, partially initialized objects. Object construction by use of the creation stack proceeds as follows. A simple object is constructed by first pushing its command template onto the creation stack, then creating the object, and last initializing its contents. When initialization is completed, the creation stack is popped and the command template is discarded. A complex object is constructed by pushing its command template onto the creation stack, creating the complex object, and then constructing its components by interpretation of the template's initialization commands. Component construction consists of first pushing the component's command template onto the creation stack and then creating and initializing the component. Since the complex object's command template is not popped off the creation stack until the object has been fully initialized, command template processing is nested. Infinite nesting depths cannot occur because the TASK compiler prevents circularities in object instantiation.

The creation stack is an appropriate data structure for the creation of trees of objects. Furthermore, it is straightforward to resolve the formal parameters associated with actual parameters at lower nesting depths in the creation stack; the loader simply accesses the command template and the associated object in the creation stack. However, a different data structure is required to resolve cross references between objects not initialized in the appropriate order and between separately constructed modules (see the description of the loader's *library* mechanism in Appendix 4 and in the internal documentation [103, 154]).

2.5.3. The Dual Role of the Loader

In the current implementation of the TASK loader, objects are constructed statically. The dynamic construction of objects requires the execution of direct calls to the operating system by the algorithmic code. Both kinds of object construction are not equally straightforward. In the case of statically constructed objects, programmers need only state the small amounts of information required by TASK, and the loader's interactions with the operating system's memory manager are transparent. In the case of dynamically constructed objects, programmers must code the protocol of interaction with the memory manager, and they must explicitly supply the parameters required for object construction. However, there are benefits to both static and dynamic object construction, and the exploitation of such benefits would be facilitated if both static and dynamic construction were equally straightforward. For example, programmers may trade the efficiency in time of once constructing an object statically against the efficiency in space of constructing an object dynamically whenever it is needed.

The TASK loader has been extended such that dynamic and static object construction are equally straightforward in the specific case of constructing processes. Processes were chosen since their substantial resource requirements during execution suggest that they should be destroyed after performing the services for which they were constructed. Specifically, as with the static instantiation of a process from a function template, dynamic process creation (but not initialization) is performed by the *invocation* [149, 80] of a function in an accessible module. In either case, construction detail concerning the process need not be specified because due to the availability of function templates. In the dynamic case, function templates are stored in the data parts of the functions' modules. When a function is invoked, the loader accesses the appropriate function template to construct a partially initialized process that contains a private stack object and an "initial code" object. This process is initialized by executing the instructions contained in the "initial code" object[8].

The loader's roles in static and dynamic process creation are illustrated in figure 2-11. The chosen

---

[8]Currently, "initial code" is written by programmers. To generate the "initial code" automatically is a straightforward extension of TASK.

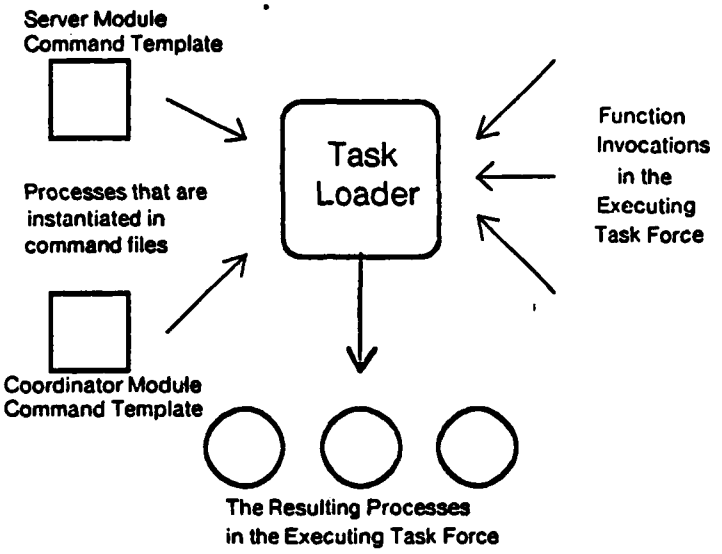


Figure 2-11: <Small Change required>The Dual Role of the Task Loader

example is a PDE task force in which additional server processes are created at runtime. We display the server and coordinator module command templates because they each contain process creation commands. In addition, the process creation commands issued by the PDE task force during its execution are displayed.

As with process construction, the construction of any object could be performed dynamically if its command template were retained after loading. For example, the command templates of each module's components could be contained in a special module component, called a *template object*. Any dynamic component construction command could reference its specific command template by the integer name ascribed to the component by TASK.

We note that the loader uses its ability to invoke functions in modules for static process construction. Specifically, to construct a process statically the loader invokes the requisite function of a module specified in the TASK program. A partially initialized process is created as a result of this invocation. Process initialization is completed by interpretation of the process' function template. In this case, the "initial code" in the default process is not executed.

2.5.4. Discussion of the Loader

In this section, we consider the performance of the loader and we illustrate the loader's usefulness by discussing the shortcomings of an earlier loader version. First, consider the storage space required for loading a task force. Although task forces are loaded one module at a time, the command file of each module is potentially large. However, the address space limitation of the loader[9] dictate that command files be kept as small as possible. Consequently, the TASK compiler compacts each command file by ensuring that only a single copy of identical object attributes or of identical templates is generated.

Next, consider the time required for loading a task force. We are not concerned with the optimization of this time because most task forces execute for considerable lengths of time. Therefore, no attempts are made to optimize the order in which objects are constructed. Typically, objects are created and initialized in the same order in which they are instantiated in TASK programs. An exception to this rule is the case in which actual parameters remain undefined at load-time. In this case, objects are created, partially initialized, and re-initialized later.

The shortcomings of an early version of the TASK loader are an appropriate means of demonstrating the usefulness of the current loader. In the early loader [78, 117], the code, the data, and the command templates of a module were coalesced into a single file in which each command template was immediately followed by the code or data of the associated object. Two shortcomings resulted from this scheme. First, objects had to be created and initialized in the order in which commands appeared because non-sequential command interpretation would have required random access to large amounts of information. Second, a created object had to be initialized with the data following its command template. Since initialization had to be completed before the next object could be constructed, repeated and partial initialization were impossible.

---

[9] Address space is extremely limited in Cm* since the LSI-11 processors are 16 bit machines.

2.5.5. Debugging and Monitoring

Although the automatic linkage of a debugger into executing software is standard practice in programming environments [62, 113], the current TASK system does not support the automatic linkage of the STAROS system's debugger [143] into a task force [48, 109]. In addition, TASK does not generate the debugger's symbol table. Instead, programmers must explicitly generate symbol table information and store it in task force components created for that purpose.

The interface of TASK to software monitoring facilities is equally rudimentary. Again, programmers must explicitly create and manipulate objects that contain monitoring information. An improved interface to a software monitor [144] is currently being designed.

2.6. Loading in a Distributed System

Loading cannot be performed without knowledge of the resources in the execution environment. Furthermore, the allocation of specific resources to task force components cannot be performed without knowledge of resource names. The subjects of this section are the resource descriptions used by the TASK loader and the binding of names to specific hardware resources. Toward this end, we first describe the Cm* hardware.

2.6.1. Distributed Hardware

Distributed hardware is a collection of processors, memories, I/O devices, and connecting busses [44, 74], where components and busses may be inhomogeneous, asymmetric, physically distributed, or inhomogeneously accessible [84]. Some examples serve to explain inhomogeneities and asymmetries. Two processors with different wordlengths are inhomogeneous, whereas two processors with identical wordlengths and different cycle speeds are homogeneous, but asymmetric. Two computers are inhomogeneously accessible if one is accessible via a fast fiber optic network link, whereas the other is accessible via a slow, telephonic link.

The scope of our investigation is narrowed by several simplifying assumptions. First, we do not

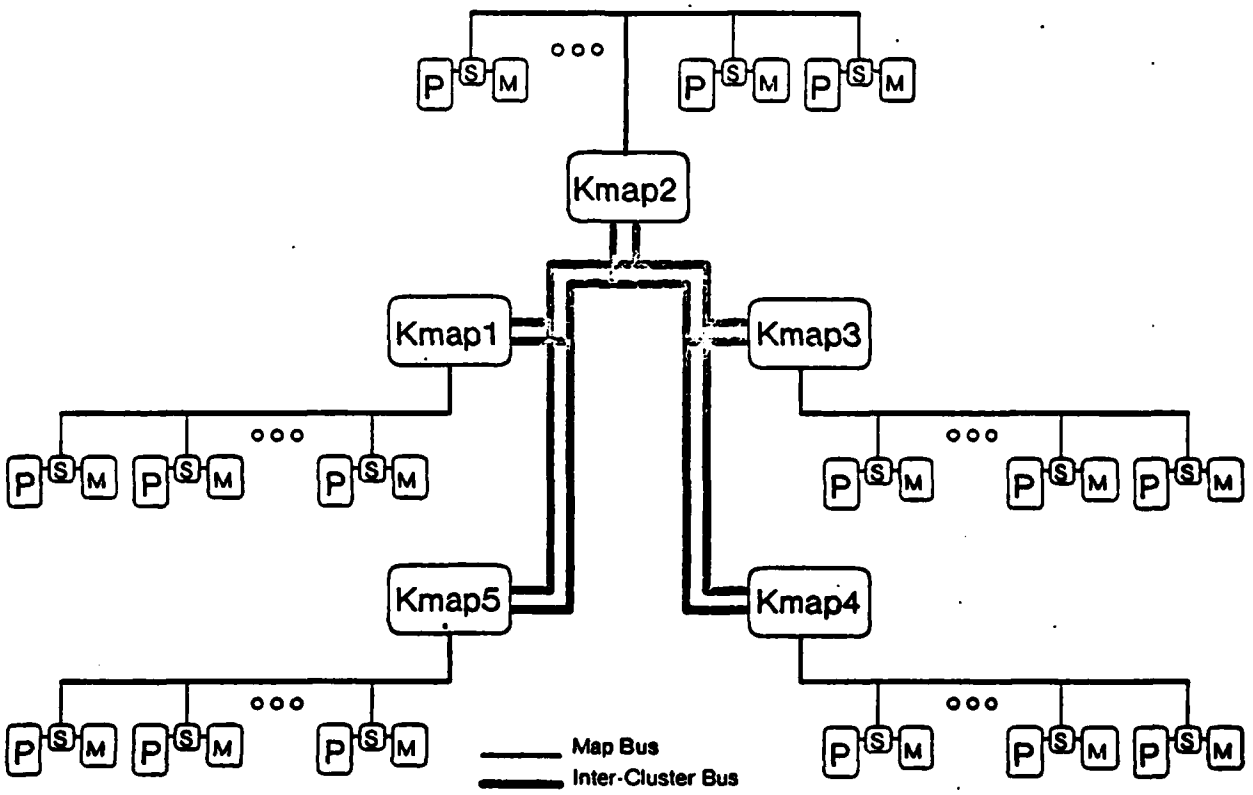


Figure 2-12: The Cm* Multiprocessor

investigate issues that pertain to component inhomogeneities, such as program portability [24]. As a result, it is assumed that all processors, memories, and busses within TASK's distributed execution environment are functionally identical, although their performance characteristics may differ. Consequently, any process can execute on any processor in the distributed execution environment, and any object can be represented within any memory unit. In addition, we are not concerned with components that are inhomogeneously accessible due to differences in communication protocols [66]. Therefore, another assumption of our research is that uniform protocols are employed for communication within the entire system.

The following hardware characteristics are considered in our investigation: memory sizes, bus

bandwidths, memory and processor reliability, and the topology of component interconnections [56]. Components are assumed logically fully interconnected at all times. Specifically, it is assumed that some physical interconnections always remain unbroken so that a change in interconnections due to temporary or permanent failures [56, 141] only effects differences in access costs. Clearly, this is a realistic assumption for multiprocessors [57]. However, the assumption is also realistic for certain network architectures. For example, although the failure of one imp (interface message processor) in the Arpanet changes the interconnection topology of the network, message routing techniques preserve the logically full interconnection of the network, albeit at the expense of message delivery times.

The specific hardware of concern is the Cm* architecture [80], as illustrated by figure 2-12 (additional detail can be found in papers by Fuller and Swan, et al. [57, 153]). The Cm* multiprocessor is composed of *computer modules*, each consisting of a DEC LSI-11, a standard LSI-11 bus, memory, and devices. In addition, each computer module includes a local switch, the *Slocal*, which routes processor memory references either to the local memory of the computer module or else onto the *map bus*. An Slocal also accepts references to its computer module's local memory that emanate from distant processors.

A *map bus* connects up to fourteen computer modules. Each map bus is supervised by a single *Kmap* processor responsible for routing data and memory requests between Slocals. Computer modules, a Kmap, and a map bus comprise a *cluster*. Cm* can include any number of clusters, which are connected via *intercluster busses*. Currently, Cm* consists of five clusters, 50 computer modules, and up to 128K words of primary memory per computer module.

The Kmaps collectively mediate each memory reference placed on a map bus, thereby sustaining the appearance of a single large memory. However, due to the cluster structure of Cm*, memory is organized in a performance hierarchy; approximate inter-reference times for local, intracluster, and intercluster references are 3, 9, and 26 micro-seconds, respectively, as measured in benchmark

tests [28].

The organization of busses qualifies Cm* as a hybrid architecture. On the one hand, it is a multiprocessor because all memory is directly accessible from any computer module in the system. On the other hand, the switching architecture is physically implemented as a network of busses.

2.6.2. Configuration Descriptions

The resource descriptions in the TASK compiler and loader are called *configuration descriptions* because they describe the Cm* hardware as configured by the STAROS operating system. Certain resources are not contained in this configuration. Specifically, we exclude the physically available resources that are not acquired by the operating system during its initialization, the resources that fail during system execution, and the resources dedicated to the operating system. For example, when the operating system is initialized, it may only use two of the five clusters of Cm*. Furthermore, if memory units are unavailable due to temporary or permanent failures [141], they are either not included into the operating system's pool of available memory, or they are excluded from the pool while the operating system is running [149, 80]. Devices like terminal lines, disks, and network interfaces can be added to or removed from the computer modules to which they are attached.

Within the TASK compiler and loader, configuration descriptions are described as collections of templates analogous to those used for software. For example, a single Cm* cluster is described by a complex template recording the computer modules attached to the cluster's map bus and a few cluster attributes, the latter including the cluster's mean time to failure, the devices attached to the map bus, and some summary statistics concerning the cluster's modules. Computer modules are described by simple templates listing attributes such as the absolute memory size, the amount of memory that remains unused, the computer module's mean time to failure, and the list of devices attached to the computer module. In figure 2-13, we graphically display a template describing a Cm* cluster with three computer modules, called Cm11, Cm12, and Cm13. The lines represent the *component of* relations between the cluster and its computer modules.

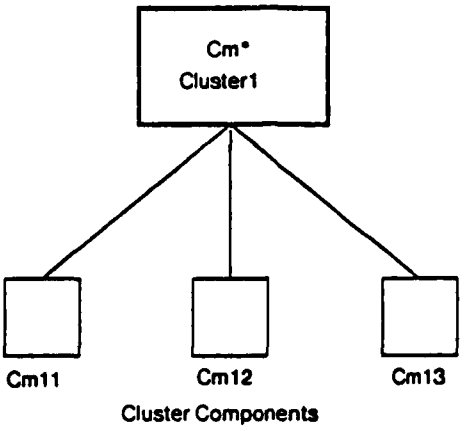


Figure 2-13: The Components of a Small Cm* Cluster

Since Cm* and STAROS change infrequently [29, 28], it could be assumed that accurate configuration descriptions are maintained by both the TASK compiler and the loader. Furthermore, the configuration descriptions that are used to make resource allocation decisions could be assumed identical to the configuration descriptions that are used when these decisions are carried out. Instead, task force construction is implemented such that construction failures will not occur when the configuration descriptions used at compile-time are inaccurate. This failsafe behavior is achieved by the compile-time allocation of virtual rather than physical hardware components. To determine the bindings of virtual to physical components, the TASK loader ascertains the accuracy of its own configuration descriptions by negotiation with the memory manager and process scheduler. It is assumed that these resource managers supply accurate configuration information. Note that this assumption is not made in other distributed system research [75, 93, 59].

Since configuration descriptions are subject to change, they are not "wired-into" the TASK compiler. Instead, they are stored in files read prior to each compiler run. Although we designed a dialect of the TASK language in which hardware configurations could be formulated [82], the TASK compiler

was not extended to read such configuration descriptions. For expediency in implementation, description formats requiring little parsing and semantic processing are used [154]. We note that an interesting extension of the TASK system is one in which the particular configuration a task force *requires for execution* is determined by TASK and set up prior to loading. In such an extension, a task force requiring sole use of a configuration of Cm* could be furnished a configuration of minimal size. As a result, the maximal number of remaining hardware components could simultaneously be used for other purposes.

2.7. TASK-Summary, Discussion, and Extensions

In this section, each of the three design goals of TASK is stated followed by a discussion of the manner in which it is attained. Next, we discuss the benefits of separating the design and implementation of the TASK language from the algorithmic code. Last, the current use of the TASK system and several extensions are presented.

The first design goal of TASK is to facilitate the variation of the executable task forces constructed from a TASK program. This goal is attained for several reasons. First, the specification of a template is *separated* from its instantiation so that a template can be instantiated repeatedly. As a result, task forces with different numbers of instantiated objects are easily constructed. Since processes are instantiated in the same manner as other types of objects, TASK compares favorably to concurrent programming languages [50, 68, 14, 15] in which the specification of a process is synonymous with its instantiation. Second, an explicit language construct is provided to permit the instantiation of multiple objects from the same template, where the values of the variables that control replication or partitioning can vary. To facilitate variation of replication or partitioning variables and to vary the values of object attributes, TASK program variables can be declared template parameters. The usefulness of template parameters in the variation of task force construction is discussed next.

With respect to the variation of construction, it would be ideal if parameters would remain unbound until task force execution. In that case, the change of a parameter value would not require re-

compilation, re-linking, or re-loading. In TASK, for expediency of implementation most template parameters are bound during task force compilation. However, to demonstrate the feasibility of delaying parameter binding times, replication and partitioning variables are not bound until load-time. Typically, these load-time variables are used to replicate objects depending on the number of resources that are available when the task force is loaded, and to partition data depending on its size.

In the TASK tool system, parameter binding times are determined by the tools involved in task force construction. For example, parameters concerning data partitioning cannot be bound by the TASK compiler since only the linker can ascertain total data size. Furthermore, virtual resource names cannot be bound to the names of physical resources before load-time since the loader negotiates virtual to physical resource mappings. Note that assumptions of bound parameters will exist in any tool system that integrates existing tools. However, such assumptions can be avoided if the components of the tool system are written to suit the functionality that is required [62, 165].

The notion of TASK program *blueprints* is developed to implement the straightforward variation of construction within the TASK compiler. In the logical blueprint, structural information is described that typically remains unchanged across experiments with the executable task force. The blueprint that varies from one experiment to another is the execution blueprint. Additional uses of blueprints have been identified. For example, blueprints can be used to change task force construction parameters without requiring the recompilation of the associated TASK program. Furthermore, as suggested at the end of this section, blueprints can aid programmers in visualizing the structures of their task forces.

The second design goal of TASK is to relieve programmers of the specification of construction detail. This goal is attained by the automatic derivation of construction detail from information specified by the programmer and by use of construction defaults. For example, the size of an object can automatically be determined by the linker if this object contains algorithmic code. Alternatively, the compiler can determine that some standard size be used for an object instantiated in a TASK

program.

The third design goal of TASK is the incremental specification and construction of a task force. This goal is attained because TASK programs contain separately specified modules that are individually linked, transferred to Cm*, and loaded. More generally, the attainment of this goal is possible since both TASK and STAROS support units of packaging that are easily manipulated. Specifically, a TASK template is a package for the specification of a task force component, and a STAROS object is a package for the runtime representation of a component. Both kinds of packages are accessed by name, where name translations are transparent to programmers. Due to this transparency, the incremental, static or dynamic construction of any object specified by a template is easily implemented, provided that the number of cross-references between the separately constructed objects remains small.

There are benefits to developing the TASK language separately from Bliss-11. Specifically, the TASK language exhibits little complexity, and the efficiency of code generation in Bliss-11 is retained. Furthermore, Bliss-11 code written prior to TASK remains compatible with TASK-generated software. The advantages of compatible languages and of languages that exhibit little complexity suggest that the extension of one language by another is preferable to the implementation of a joint language, such as Concurrent Pascal or Ada. However, one must consider whether the manner in which two languages can interact will limit their joint functionality. For example, since only the file names of algorithmic code are known in TASK programs, the TASK compiler cannot determine the specific code and data that constitute object contents. As a result, the compiler cannot automatically replace a TTY driver by a screen-oriented code module when a screen device becomes available in Cm*. Similarly, the compiler cannot automatically replace code that implements process communication via shared objects by code that implements communication by messages. Such automatic selection of code would either require a joint implementation of both languages or it would require an interface of TASK to Bliss-11 in which the semantics of code procedures are known. We hypothesize that such an interface would be sufficiently complex to warrant the joint implementation of both languages.

However, a less detailed interface in which TASK programs contain Bliss procedure names and parameters appears straightforward to implement. In this case, interface and version control [62] could be implemented in TASK.

Currently, TASK is being used to construct experimental software for the Cm* multiprocessor. Such software tends to be static; it does not experience frequent changes in resource usage during execution. Therefore, this experimental software can almost entirely be constructed by TASK so that programmers need not be concerned with the use of operating system facilities. In addition, Cm* software is typically constructed once and run repeatedly. As a result, task force construction need not be efficient in time. One might suspect that most multiple processor applications in the "real world" differ from Cm*'s experimental software, thereby limiting the usefulness of a system like TASK. This is not the case. For example, most real time software [17] does not require runtime resource allocation mechanisms. Similarly, image processing programs [21] are usually initialized once and subsequently run repeatedly for multiple images. Other applications similar to Cm* software are: (1) signal processing [136, 12] and speech recognition [106, 31] programs, (2) traffic control [16] programs, and (3) multiple process file servers or network message servers that are instantiated once to serve recurring service requests.

Two possible criticisms of the TASK system are that TASK is inherently static, whereby its usefulness is limited, and that graphical design tools are required to aid programmers in the development of the complex structures of task forces. To disspell the notion that TASK is inherently static, we have extended the TASK language so that task forces that exhibit frequent, dynamic change can be constructed. Specifically, each object described in a TASK program can either be constructed by use of direct operating system calls [80] or statically constructed by use of the loader. This extension has proven useful. Typically, programmers will initially use TASK to construct all objects in the task force. However, as they become more familiar with the system and once their task forces are debugged, the programmers' concerns with task force performance can prompt the dynamic construction of selected task force objects under program control.

A dynamically constructed object is a component of a task force which is described in a TASK program. To allow such an object to be named and manipulated like other task force components, the object is declared in the TASK program to the degree necessary. Dynamically constructed objects can be declared in two ways, as **Reserved** or as **Named** objects. In the case of **Reserved** objects, the TASK system attempts to ensure that the resources required for object construction will be available at the time of construction. Specifically, the resources required by **Reserved** objects are considered part of the total resource requirements of a task force, and task force construction will not complete without warnings if total resource requirements are not met by the Cm* configuration used for loading. The other way of declaring a dynamically constructed object consists of providing an object **Name**, where neither the type of the object nor any of its attributes need be stated.

Dynamic object construction performed by the algorithmic code of a task force would be facilitated if the loader's command templates were available at runtime. Although this is not the case in the current implementation of TASK, we have demonstrated feasibility by implementing the dynamic construction of processes. Process construction templates are stored in module objects.

For complex task forces, development aids in addition to the TASK language may be required. For example, if many cross-references exist between components at different levels of the tree representing the task force, then graphical displays are required to visualize the task force structure described by the linear prose in a TASK program [45, 46]. Note that the structural characteristics mentioned here are described in the logical and execution blueprints of a task force. Therefore, graphical representation of these blueprints are the additional aids that are required by programmers. Furthermore, graphical displays displays of blueprints could present menus of task force components that can be selected for inclusion into the blueprint displayed. For example, given a logical blueprint, programmers could manipulate its display to derive an execution blueprint that contains an object with code that implements a specific process communication scheme. In this fashion, the construction of different executable task forces could be partially automated. Alternatively, component selections could be transparent to programmers, the feasibility of which has been shown

elsewhere [140, 159, 11, 105, 122].

3. Task Force Tailoring

3.1. Overview

In the previous chapter, we discussed the specification and construction of a task force. The subject of this chapter is the allocation of physical resources to the components of a task force. *Tailoring* is defined as the set of resource allocation decisions concerning task force components. Since tailoring is tedious and is difficult to perform for task forces containing more than a few components, it is our objective to automate tailoring to the greatest extent possible. Therefore, we will identify the information required to perform task force tailoring, and we will express this information with sufficient precision to permit automation.

Attainment of automation is dependent on several factors. First, it is difficult to obtain precise information at any particular instant of time concerning the distributed hardware on which a task force executes. Therefore, tailoring decisions may be based on incomplete or imprecise knowledge. Second, multiple objectives that are pursued during tailoring may result in conflicts, whereby a good decision with respect to one objective is a poor decision with respect to another. For example, consider two tailoring objectives: improving reliability in execution and speedup of execution. To improve reliability in execution, the use of duplicate or backup components is suggested. However, the maintenance of such additional components is likely to reduce a task force's speed of execution.

In the following, our definition of tailoring is formalized and refined so that tailoring objectives can easily be varied. The target hardware information that must be available for tailoring is identified and is assumed to be known. We present examples of task force tailoring on Cm* to show that the chosen formalization is meaningful in practice. In addition, it is demonstrated that the formalization also applies on architectures other than Cm*. Furthermore, the two previously mentioned tailoring objectives are considered: increasing task force speedup and increasing reliability. While speedup tailoring is discussed in detail, reliability tailoring is only touched upon to show the manner in which

different tailoring objectives are formulated. In all examples, task forces are tailored individually so that interactions between multiple, executing task forces are not considered.

3.2. The Proximity Model

Since tailoring depends on the task force and the target hardware, we define a model in which both can be expressed. In the *proximity model*, information is formulated as objects related by proximity relations. *Proximity* is interpreted broadly. In one case, objects represent hardware components and proximity corresponds to the physical distance between components. In another case, objects represent task force components and proximity corresponds to the frequency with which one object accesses another. In an *instance* of the proximity model, we state the interpretation of proximity by defining a set of binary relations, where each relation can specify the proximity of two objects. In addition, an instance defines a set of objects and properties characterizing each object. The following specification is a sample abstract instance of the proximity model in which properties do not appear:

Objects: {1,2}
Relations: {*Diff*(1,2)}, where *Diff* expresses the difference between 1 and 2

The software and hardware instances that will later be used for speedup tailoring on Cm* are further examples of model instances. The hardware instance represents the Cm* architecture. The computer modules are the objects of this instance, and the proximity of computer modules corresponds to physical bus lengths. Three relation values are of interest in Cm*: (1) if *Same-Cm* holds, two components are connected by the LSI-11 bus, (2) if *Same-Cluster* holds, two components are connected by the intracluster bus, and (3) if *Different-Cluster* holds, two components are connected by one or more intercluster busses. Consider an instance describing a two-cluster Cm* configuration, where Cm_{ij} denotes the j-th computer module in cluster i.

Objects: {Cm11, Cm12, Cm13, . . ., Cm19, Cm21, Cm22, Cm23, . . ., Cm29}
Relations: {*Same-Cm*(Cm11,Cm11), *Same-Cluster*(Cm11,Cm12),
Same-Cluster(Cm11,Cm15), *Different-Cluster*(Cm11,Cm22)}

The relation *Same-Cm*(Cm11,Cm11) is used to state the physical distance of a computer module to

itself, where this distance corresponds to the length of the module's LSI-11 bus.

The definition of a model instance does not require that the relations be complete. For example, the Cm* description above is incomplete since a complete description requires listings of the physical bus connections between each pair of computer modules. We also permit inconsistent sets of relations. For example, the addition of the relation *Same-Cluster*(Cm11,Cm22) to the hardware instance above makes the relations inconsistent because the added relation conflicts with the relation *Different-Cluster*(Cm11,Cm22).

The sample software instance of the proximity model describes a task force. The *objects* of that instance are the processes $p_1,...,p_n$ and the code, stack, and mailbox[10] objects $o_1,...,o_m$ of the executing task force. We attach a *type* property to each object to distinguish between *active* objects of type *process* and *passive* objects which have other type property values. Given this distinction, we define relations between p_k and o_i, o_i and o_j, and p_k and p_l. A relation between a process p_k and a passive object o_i is defined as the *frequency of access* of the process to the passive object. Two examples of *frequency of access* relations are: (1) a process frequently accessing its code and stack objects, and (2) a process rarely accessing the object it uses to store final computation results.

A *potential access* relation[11] between two passive objects o_i and o_j is defined to express the implication of a *frequency of access* relation between a arbitrary process p_k and o_i with respect to p_k's frequency of access to o_j (or vice versa). For example, if the value of a potential access relation between two code and stack objects is "frequent", then any process that frequently accesses the code object will also frequently access the stack (and vice versa). However, since the access of a process p_k to its own stack does not have any implications concerning the access of p_k to the stack of a different process, a "null" potential access relation should be stated between the two stacks.

---

[10] Mailboxes are used for process communication [149].

[11] Potential access relations are useful when specifying proximities within TASK programs (see Chapter 4).

Constraint relations between passive objects o_i and o_j or active objects p_k and p_l are defined to restrict resource allocation. For example, a constraint relation between two processes could either state that they must execute in parallel or state that they may share the use of one processor. Constraint relations will be discussed in detail in Chapter 4.

To summarize, the following information is specified in a sample software instance of the proximity model:

Objects: {the objects of the executing task force}
Relations: {{*frequency of access*: active objects-passive objects,
passive objects-active objects},
{*potential access*: passive objects-passive objects},
{*constraints*: passive objects-passive objects,
active objects-active objects}}

We have defined the proximity model in order to provide a basis for the automation of tailoring. Toward that end, an operational definition of tailoring is provided. Given a software instance and a hardware instance, *tailoring* a task force consists of specifying a surjective mapping of the objects in the software instance to the objects in the hardware instance. Tailoring is performed to achieve some desired objective, such as balancing bus usage in a system with multiple busses. In order to measure how well an objective is achieved by a mapping, we define a *metric function* that ascribes a value to the mapping. A metric function is chosen based on the proximity model instances involved and upon the tailoring objective being evaluated. For example, to evaluate tailoring with respect to balancing bus usage, a possible metric function computes the total load on the busses due to the access frequencies between task force objects. In this case, a good tailoring decision corresponds to a mapping with a small metric function value. Small values are achieved when (1) task force objects related as "less-frequent" are placed into modules residing in different clusters (modules related by the relation *Different-Cluster*), and (2) task force objects related as "frequent" are placed into modules residing in the same cluster (the relation *Same-Cluster*).

The presented metric function is inappropriate in practice because it does not express that certain mappings are infeasible due to physical constraints. A sample infeasible mapping is one that places

passive objects into a computer module in excess of the module's limited amount of memory space. Metric functions that eliminate infeasible mappings take *properties* of objects into account. Typical examples of useful properties are the *size* of a task force object's representation and the *size* of the available memory in a computer module. An example of a feasible mapping is one that assigns all processes in a task force to execute on a single processor to limit the total load on busses. However, this mapping is inappropriate with respect to the objective of realizing a task force's potential parallelism. To avoid choosing this inappropriate mapping, a metric function can be designed to emphasize the constraint relations that state the desired parallelism within a task force. The use of such metric functions during tailoring will prevent the assignment of several processes to a single computer module (see Chapter 5).

To summarize, the presented tailoring definition enables us to quantify tailoring with respect to a stated objective. While better or worse tailoring decisions correspond to low or high metric function values of the mappings of the software to hardware instances, the resulting quality of such tailoring decisions is dependent both on the precision of the available proximity relations and the appropriateness of the metric function used for tailoring. This topic is discussed further in Chapter 5.

3.3. Tailoring Objectives, Metric Functions, and Proximity Relations

In this section, we first discuss the manner in which metric functions are associated with tailoring objectives. We then present the *speedup* tailoring objective and the associated metric function, proximity relations, and object properties.

The association of metric functions with tailoring objectives is not always straightforward. Specifically, multiple metric functions can be associated with one objective, and one metric function can be associated with several objectives. As a sample tailoring objective, consider the elimination of bottlenecks in an existing system. Two different metric functions can be used to evaluate tailoring with respect to this objective: system throughput and system utilization. System throughput, as formulated

in Kleinrock [86], is a suitable metric function because throughput is improved if and only if bottlenecks are eliminated. System utilization is a suitable metric function because bottlenecks are typically caused by system components that are over-utilized. However, the metric function system utilization can also be used with another objective: minimizing the cost of hardware while it is being designed.

In this thesis, the tailoring objective is to improve the performance of a single task force that uses multiple processor and memory resources. We equate the performance of a task force with the speedup of task force execution gained by parallelism. *Speedup* is defined as the ratio between a task force's elapsed time to completion with one process and the same task force's elapsed time to completion with "n" processes. For example, speedup equals ten if a task force's computation is completed ten times faster with ten processes than with one process.

Speedup depends upon (1) the ability of task force processes to execute in parallel and (2) the elapsed times to completion of each process. We are not concerned with (1) because the ability to execute in parallel is usually determined by the control flow within the executing task force. Regarding (2), consider a sample task force in which all processes execute in parallel. In this case, task force completion time directly depends upon process completion times; task force completion time is equal to the maximum of the completion times of task force processes. The completion time of each process depends upon what we call *communication time*, the amount of time the process spends in accessing code and data. In the example above, the manner in which task force completion time depends upon process completion times and therefore, upon process communication times is straightforward. Although this dependency is not always as simple, we employ total communication time as a metric of speedup tailoring. A task force's total communication time is computed by adding the communication times of its processes.

We next demonstrate that total communication time is an appropriate metric with respect to the proximity relations in the previously defined software and hardware model instances. An analogy to

the well-known working set model [38] in paging systems supports our reasoning. In the working set model, the dynamic working set of a process is defined as the set of recently referenced pages. Paging algorithms minimize the number of page faults during execution by keeping the dynamic working set of each process in primary memory. As a result, process completion times are reduced. In our analogy, the static working set of a process is the set of objects frequently accessed. This set is determined by the *frequency of access* relations in the software instance. Speedup tailoring algorithms attempt to minimize communication time of each process by locating each process' static working set near the site of process execution so that the objects are rapidly accessible. As a result, total communication time and task force speedup are improved.

We conclude by noting that the communication time metric is a commonly used evaluation criterion in the experimental and analytical literature. Specifically, measures of communication time have been used (1) in experimentation with the Cm* and Cmmp multiprocessors [29, 166, 111], (2) in analyses of alternative network designs [56], (3) in analyses of distributed data bases [18], and (4) in analyses of parallel algorithms [101]. Therefore, we are able to compare our work with related research (see Section 3.10 and Chapter 5).

3.4. The Execution Environment

In this section, we discuss the environment for which a task force is tailored. Recall that this environment consists of hardware components that are configured by the operating system (see Chapter 2, Section 2.5.2). Clearly, a hardware instance of the proximity model must record the configuration of hardware so that configuration can be taken into account for tailoring. For example, a process cannot be assigned to a processor dedicated to an operating system server process, and an object cannot be placed into a memory totally occupied by operating system data. We refine the previously defined hardware instance by recording two different kinds of objects: processors and memory units. For each memory unit, we record its physical size (`MpSize` in table 3-1) and the amount of memory augment the previously defined hardware instance by recording two different

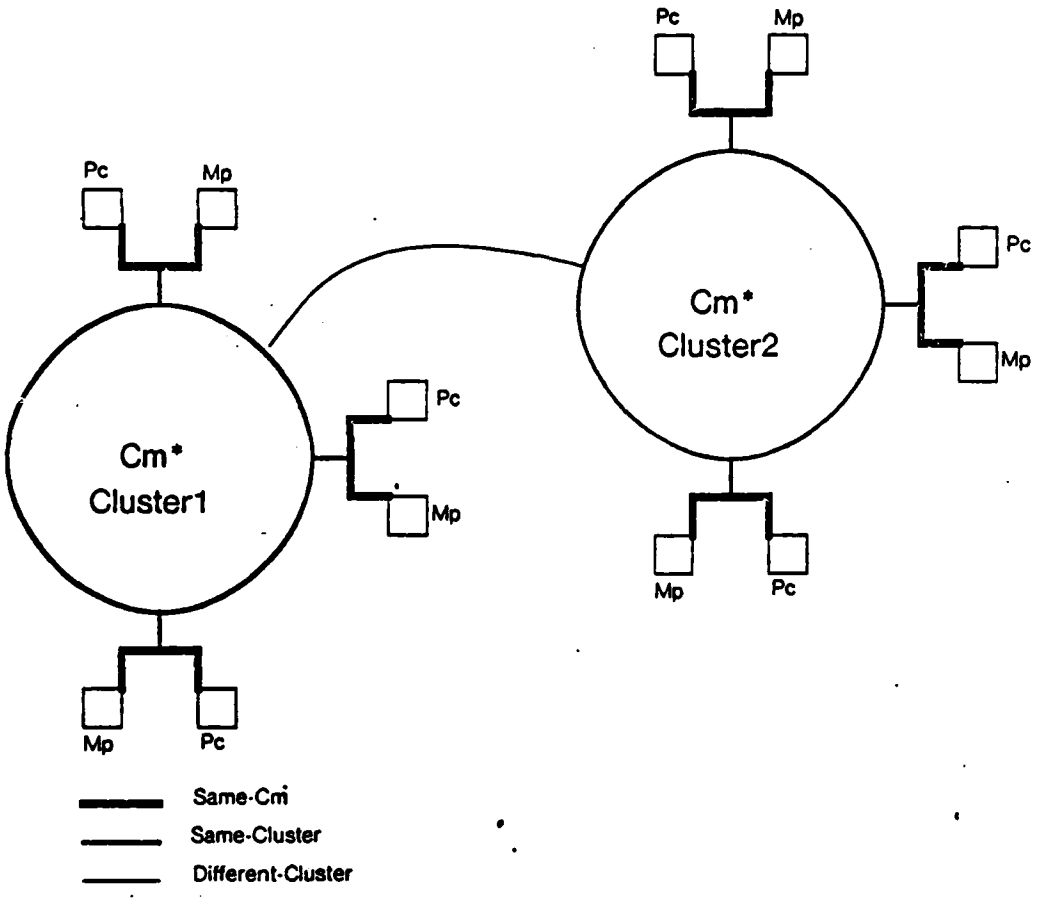


Figure 3-1: The Proximity Relations of two Cm* Clusters

kinds of objects: processors and memory units. For each memory unit, we record its physical size (`MpSize` in table 3-1) and the amount of memory augment the previously defined hardware instance by recording two different kinds of objects: processors and memory units. For each memory unit, we record its physical size (`MpSize` in table 3-1) and the amount of memory that remains unused by the operating system (`MpAvailable`). Similarly, for each processor, the physically and actually available processor cycles are recorded. In addition, specializations of hardware components are expressed.

For example, the boolean `HasDisk` is attached to processors and memory units to express whether they are directly accessible to a disk device. The boolean `HasEther` indicates whether an EtherNet link is attached to a processor or memory unit. By use of these boolean values, a disk access buffer can be placed into a memory unit directly accessible to a disk, and a network server process can execute on a processor to which an EtherNet link is attached.

The processors and memory units in Cm* are connected by three different physical links: the LSI-11 busses, the map busses, and the intercluster busses. Accordingly, the values of the hardware instance's proximity relations express the durations of accesses across these links. The rapid access of a processor to local memory (3 μ seconds) is expressed by a small value of the proximity relation *Same-Cm*. The slower accesses to memory units within a cluster (9 μ sec) and across clusters (27 μ sec) are expressed by larger values of the relations *Same-Cluster* and *Different-Cluster*. Since Cm* is logically fully interconnected, each memory unit is related to each processor. In summary, Cm* is described by the sets that follow:

Objects: {the processors and memory units of Cm*'s current configuration}

Relations: {the relations *Same-Cm*, *Same-Cluster*, or *Different-Cluster* between processor and memory units}

A sample hardware instance of two clusters of Cm* is graphically depicted in figure 3-1. Each cluster contains three computer modules. A computer module is represented by a memory/processor pair related by the relation *Same-Cm*. A circle is drawn to indicate the binary relations *Same-Cluster* between each two modules in a cluster. The relations *Different-Cluster* between the modules in different clusters are represented by one line connecting the cluster circles. Properties are elided in this figure because they are separately listed in table 3-1. We make several assumptions concerning the information in the hardware instance and the method of use of this information during tailoring. First, hardware characteristics are assumed constant between the time at which tailoring decisions are made and the time at which they are carried out. As a result, tailoring decisions need only be made once. For example, since the amount of unused memory is assumed

Table 3-1: The Properties of Hardware Components

HasEther:	Boolean indicating whether an ethernet link is connected to the computer module
HasDALink:	Boolean indicating whether a direct access link is connected to the computer module from CMU's DEC KL-10 processor
HasDisk:	Boolean indicating whether a disk is connected to the module
HasLine:	Boolean indicating whether a terminal line is attached
MPSize:	Integer encoding the amounts of physical memory attached to the computer module
MPAvailable:	Integer encoding the amount of physical memory that remains unused
PCAvailable:	Integer encoding the number of processor cycles that remain unused

constant, object placement decisions made prior to loading need not be revised by the loader. Similarly, bus bandwidths and latencies are assumed to remain constant under varying loads so that tailoring decisions made based on the proximity relations of the hardware instance are accurate. Second, each processor is assumed to be fully utilized if any one process executes on it. As a result, process assignment is simplified because processors need not be assigned in parts. Third, all processors are assumed to be of equal speed to avoid taking varying processor speeds into account. Fourth, we assume that tailoring decisions minimize the total amounts of communication within the distributed hardware to the extent necessary to avoid contention. As a result, the assumption of constant bus latencies is realistic in practice despite the observations in [29, 28].

The presented hardware instance contains the information required for speedup tailoring on Cm*. A different hardware instance will be defined when we consider reliability tailoring in Section 3.6. Similarly, alternative hardware instances would have to be developed if we were concerned with tailoring on network architectures. Consider the example of the Arpanet architecture [64, 126]. In the Arpanet, the transmission time of a packet depends upon the number of nodes the packet traverses on its path from source to destination host. Therefore, we define that proximity values between hosts

express the average path lengths of transmitted packets [18]. As another network example, consider the local Ether network [115]. Here, we define that proximity relations express average message transfer times that include the durations of message assembly and disassembly. In this case, proximity relation values reflect variations in (1) the implementation of network servers, (2) the network access protocols, and (3) the physical bus lengths.

3.5. Speedup Tailoring

In this section, we provide speedup tailoring examples in an existing experimental environment, the Cm* multiprocessor and the TASK tool system. In addition, we define the information required for tailoring in TASK.

Recall that the compilation of a TASK program results in the construction of an execution blueprint of the specified task force. Based on this blueprint, the executable task force can be constructed. It would be expedient if the task force could also be tailored based on this blueprint. That is not the case. Instead, we will show that the information in the execution blueprint is necessary but not sufficient for tailoring. As a result, we define a third task force blueprint, the proximity blueprint,

3.5.1. The Proximity Blueprint

A task force's proximity blueprint consists of some information in the execution blueprint and some information not in the execution blueprint. The *proximity blueprint* of a task force is an instance of the the proximity model. It contains the same objects as the task force's execution blueprint. However, it does not contain the *component of* relations or all of the object attributes of the execution blueprint. Instead, the relations in the proximity blueprint are the *frequency of access* and *constraint* relations of the software instance explained in Section 3.2. The recorded object attributes are *size* and *type*.

Consider a sample proximity blueprint of the PDE task force (see figure 3-2). This blueprint contains the coordinator process and its stack and code objects, three replicated server processes and their replicated stack and code objects, and the grid partitions. Proximity relations express the

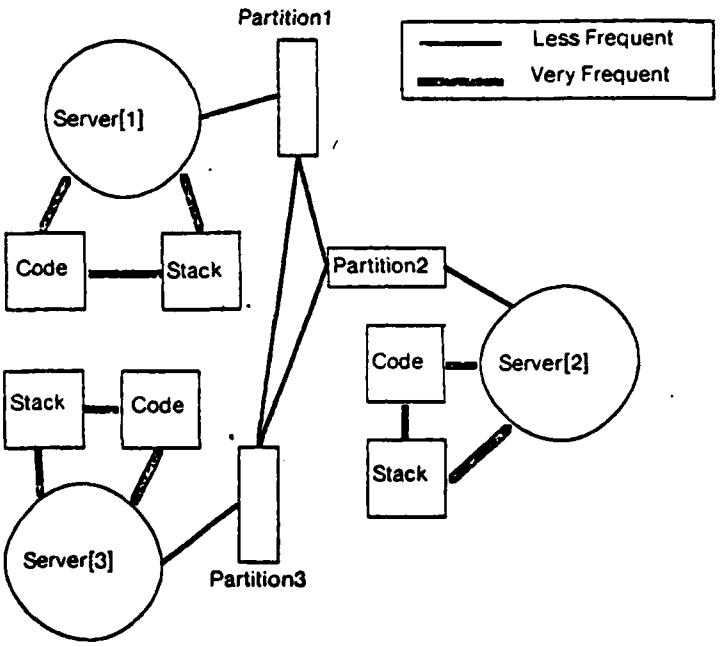


Figure 3-2: Some Proximity Relations Between Three Server Processes

access frequencies of processes to code, stack, and grid partitions, as well as the potential access frequencies between code and stack objects and between grid partitions. For simplicity, both kinds of relations are expressed in the same units. Furthermore, we only distinguish between the access relation values "frequent" and "less frequent".

3.5.2. Tailoring Examples

Deminet's experiments with the PDE task force [29] are used to illustrate that programmers typically perform tailoring experiments in which both the execution and the proximity blueprint of a task force are necessary. In each experiment, the execution blueprint is shown to be a necessary, but not

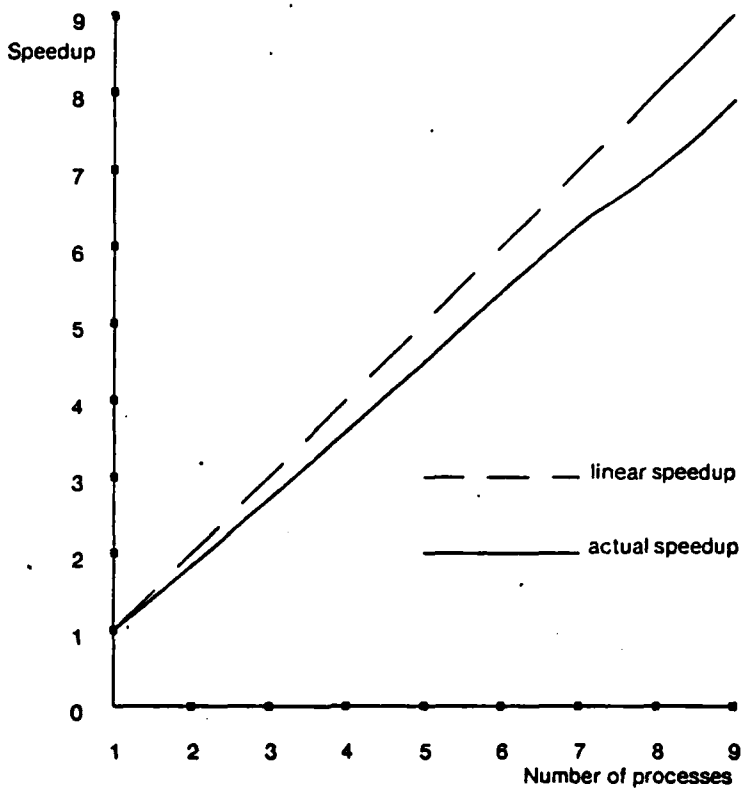


Figure 3-3: PDE-StarOS, Actual vs. Linear Speedup

sufficient basis for tailoring. The execution blueprint is necessary because the number of objects in the executable task force is determined by the values of replication and partitioning parameters. Furthermore, the values of *size* attributes are used to determine the amount of resources that must be allocated during tailoring. However, since the execution blueprint does not contain frequency of access relations, reasonable tailoring decisions cannot be made based on its contents. Instead, tailoring is performed based on the frequency of access, potential access frequency, and constraint relations contained in the proximity blueprint.

Three examples of Deminet's experiments with the PDE task force are presented. In the first example, we replicate the ..rver processes of the PDE task force in order to increase useful paral-

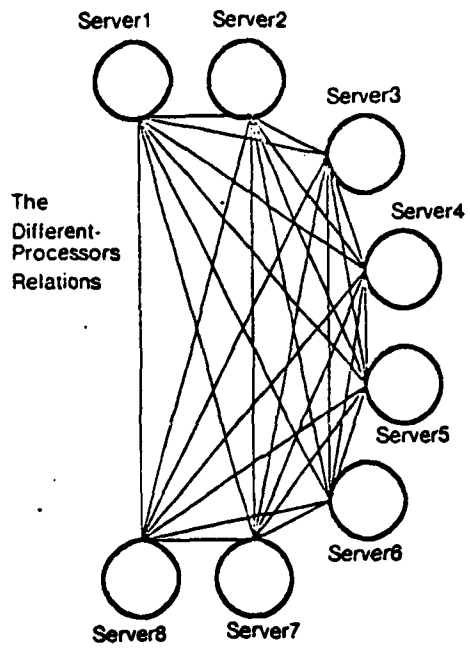


Figure 3-4: Specifying the Relations *Different-Processor* Among Eight Servers

lelism. The expectation is that server replication will improve speedup. Experiments demonstrate that this expectation can be realized if tailoring results in the assignment of each server process to a different processor. Specifically, if eight replicated server processes are each assigned to a different processor in one Cm* cluster, then speedup can be improved almost linearly, namely sevenfold (see figure 3-3).

A straightforward optimization of the speedup metric (total communication time) suggests that all processes be assigned to a single processor. In this case, speedup is not improved by server replication. Constraint relations stating that all server processes must execute on different processors are required to prevent such inappropriate assignments. We display these relations in figure 3-4. A binary relation *Different-Processor* appears between each pair of eight server processes. We note that the *component of* relations in the execution blueprint are not sufficiently rich to express these con-

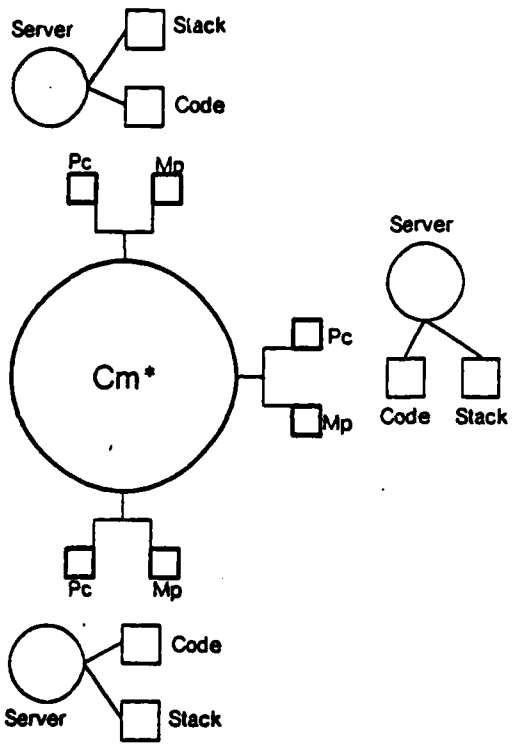


Figure 3-5: The Resources Allocated to three Server Processes and to their Objects

straints.

In practice, in addition to constraint relations, other proximity relations are required to improve speedup by process replication. For example, the access rates of processes to passive objects must be known. Specifically, since each server process frequently accesses both its stack[12] and code objects, these objects must be replicated whenever a server is replicated. Furthermore, each server process must execute on the computer module containing the server's code and stack. Otherwise, total communication within the task force is prohibitively high and causes task force speedup to

[12] Stack objects are used to store intermediate, computational results that cannot be maintained in processor registers [9].

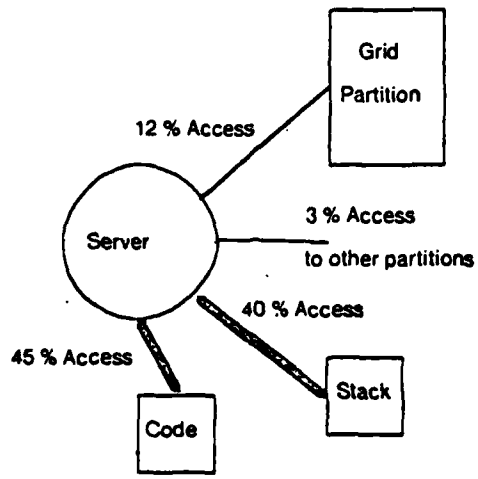


Figure 3-6: Estimated Access Rates to the Code, the Stack, and the Grid Partition

decrease rather than to increase when additional server processes are instantiated. If the code and stack are not replicated, or if code and stack placement do not concur with process assignment, then dramatic speedup decreases are observed on Cm* [29, 28] due to memory and bus contention. We illustrate the assignment and placement of three server processes on a small cluster of Cm* in figure 3-5. The software objects are drawn beside those hardware components to which they are assigned. In addition, we show the observed access rates of a server process to its code, stack, and grid partition in figure 3-6. Note that the heavier lines correspond to higher access rates, whereas thinner lines indicate less frequent access.

The example presented shows that process replication precipitates the replication of the objects in the static working set of the process. In Cm*, typically 90% of all memory references emanating from an executing process are to code and stack objects. Therefore, process replication causes the replication of code and stack. We note that objects containing data cannot be replicated if the logic of algorithmic code requires that they be shared.

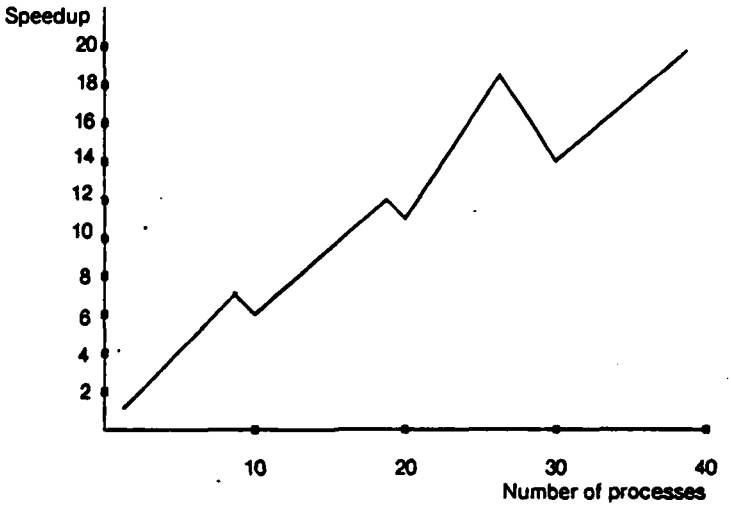


Figure 3-7: PDE-Workload Imbalances Affecting Speedup

In the second example of Deminet's experiments with the PDE task force, we illustrate that speedup is often influenced by the manner in which frequently accessed data is partitioned and distributed among processes. Consider an execution blueprint of the PDE task force in which the grid data is partitioned into a number of objects. Each server process iterates over a particular grid partition. The proximity blueprint shows that each server directs 12% of its memory accesses to a particular grid partition and 3 % to all other partitions (see figure 3-6). Given this information, the workload of each server process can be expressed as a function of (1) the size of the server's grid partition, (2) the proximity of the server to its partition in terms of access frequency, and (3) the access cost of a server to its partition. Workload differences among servers have a significant effect on speedup because

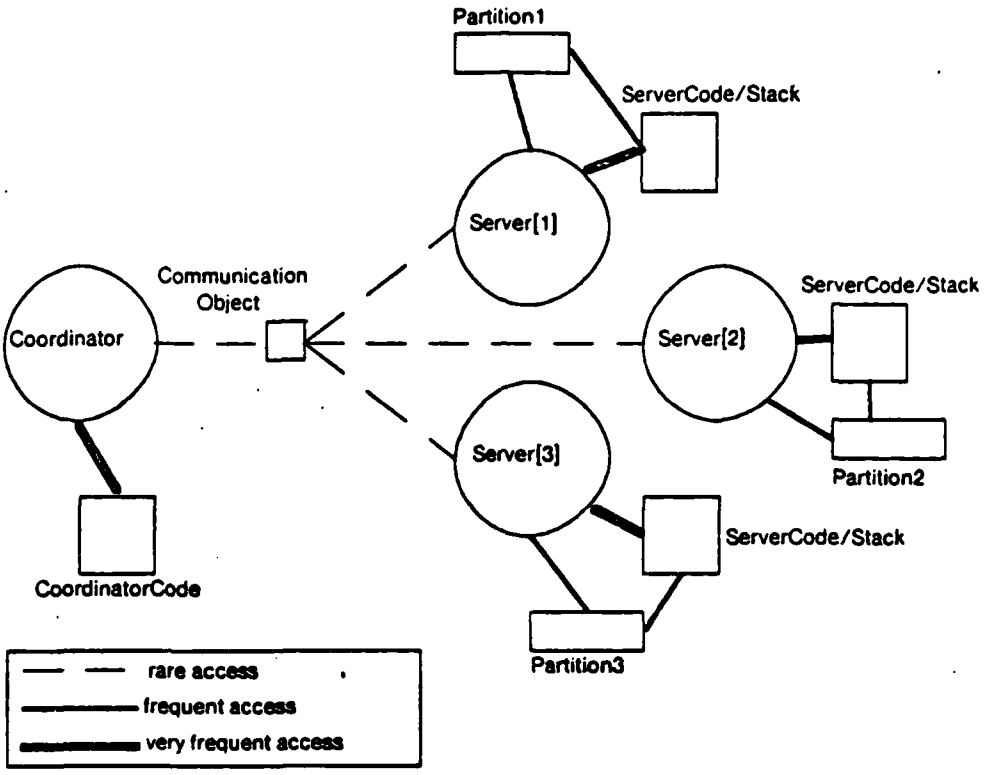


Figure 3-8: Estimated Access Rates of three Server Processes to Code and Data

PDE task force execution is not completed until the slowest process is completed. Significant speedup improvements are achieved if a grid partition distant from its server is smaller than a grid partition close to its server[13]. Alternatively, speedup is degraded if the partitions of an equally divided grid are not placed equally close to their servers. For example, if server processes execute on two clusters, then placement of the entire grid into one of the two clusters degrades rather than improves speedup. This is due to the fact that some server processes experience increased communication across cluster boundaries, thereby increasing the workloads of those servers.

---

[13]Note that tailoring can benefit from operating system support for small objects. In segmentation systems [125], only the relatively large segments and processes can be separately manipulated. In object-oriented systems, many small objects may be separately constructed, addressed, and moved. Therefore, object boundaries can be drawn to correspond to the subdivisions of code and data determined by the locality of reference patterns.

The effects of workload imbalances are demonstrated in figure 3-7 [29]. In this experiment, server processes are replicated and assigned to different processors. Servers are assigned to the minimal number of clusters possible, where clusters contain 9, 10, 9, and 9 computer modules, respectively. For example, 19 of the 20 instantiated servers execute in two clusters, and one server execute in the third cluster. Since grid data is distributed equally among clusters, a third of the grid data is placed into each cluster. As a result, a third of the grid data is close to the single server in cluster three and is distant from servers in other clusters. Since each server processes an equal fraction of the grid, workloads are imbalanced. As a result, speedup decreases whenever server replication causes the use of a new cluster. (Note the speedup values associated with 10, 20, and 30 processes.) However, as additional server processes are instantiated, workload imbalances are reduced and speedup is increased. Similar results regarding workload assignment are put forward by Gylys [61] in a theoretical study of real-time system performance.

In the third example of Deminet's experiments with the PDE task force, we illustrate that the *component of* relations in the execution blueprint are necessary but not sufficient for tailoring. Consider a server process that contains an object called **Data** which is used to store final computation results. *Component of* relations in the execution blueprint state that this object is accessible to a server process. Since object accessibility is a prerequisite for dynamic access, it is possible to conclude that **Data** is part of the server's static working set. As a result, we replicate the object, and we place it into the memory of the computer module to which the server is assigned. These actions represent poor tailoring because the **Data** object is rarely accessed by the server. Although the logic of the server's code may require the **Data** object be replicated, it may be placed wherever memory is available. While *component of* relations define the boundaries of static process working sets, these relations do not contain sufficient information to permit the determination of the access rate relations in the proximity blueprint (see figure 3-6).

A graphical representation of the entire PDE proximity blueprint is presented in figure 3-8. We display the access frequencies and the potential access frequencies discussed above. For simplicity,

one box is drawn to represent the code and the stack of each server. Potential access relations are expressed in the same units as access relations. Although the two module objects are elided, it is clear that this simple figure contains a substantial amount of information. Specifically, the relation values vary by several orders of magnitude: the code and stack objects are very frequently accessed, the grid partitions are accessed with some frequency, and the communication object is only rarely accessed.

To summarize this section, we note that speedup tailoring involves both the execution and the proximity blueprints of a task force. As demonstrated by the first example of Deminet's experiments, a simple means of achieving different speedup is to vary the values of the process replication parameters and to assign replicated processes to different processors. These assignments involve the use of constraint relations. However, in practice on Cm*, speedup improvements are gained only if the static working set of a process is replicated when process replication occurs. Furthermore, in the second example of Deminet's experiments, we show that it is necessary to balance workloads in order to attain speedup when processes are replicated. As demonstrated in the third example, although the *component of* relations in execution blueprints delimit the static working set of each process, object replication and assignment must be based on a more precise representation of the working set. The object access rates in proximity blueprints provide such precision.

3.6. A Note on Reliability

In this section, we touch upon reliability tailoring in order to demonstrate the manner in which a significantly different tailoring objective is phrased within the proximity model. Experimental results are not available[14]. To simplify, we do not identify causes for unreliability of task force components. Consequently, the following topics are outside the scope of this discussion: (1) the influence of programming techniques on reliability, such as programming for robustness in execution, (2) the

[14] Additional analysis and experimentation are required to argue that the presented reliability tailoring methodology is appropriate in practice.

relationships between reliability and object size or complexity, and (3) the relationships between reliability and the logic of the algorithmic code, such as identifying code *singularities* or *logical distributions* of code and data [84].

Given these simplifying assumptions, it is straightforward to define software and hardware model instances. In the instance that describes the task force, the reliability of each object is characterized by a probability distribution that represents mean time to failure (MTTF). Failures include errors and permanent or transient faults [20]. Similarly, we record MTTF distributions for each object in the hardware instance [141]. Due to the connections between objects, failures can propagate so that dependencies between object failure distributions can be identified [141, 20]. We define hardware and software proximity relations that express the correlations and the dependencies between object failure distributions. These relations exhibit large values .. two failure distributions are highly correlated or if they are interdependent. Otherwise, their proximity values are small.

To measure reliability tailoring decisions, we define a metric function that evaluates the match between a software and a hardware instance. For example, a poor match is a mapping in which two highly correlated task force objects are placed into two memory modules with small correlations of failure. In this case, failures within one task force object that are likely to cause failures in the other are needlessly protected from coinciding hardware failures.

Reliability tailoring is performed by searching for best matches of the presented software with the hardware instance. However, as in the case of speedup tailoring, reliability tailoring experiments [20] again involve the execution blueprint of a task force. For example, a standard way to improve reliability is to replicate selected task force objects and to place them into memory modules with uncorrelated failure distributions. The resulting redundant object copies protect task forces from irreversible information losses. Similarly, the amount of information that is lost in a failure is reduced if data is partitioned and placed into uncorrelated memory modules. Another way to improve reliability is to replicate processes that implement important services (e.g. network servers) and to

assign them to uncorrelated processors. As a result, the availability of the important services is improved [84].

We note that reliability tailoring may not be possible without altering the algorithmic code of a task force [29]. For example, algorithms that require access to shared data may have to be altered if such data is to be replicated. Similarly, algorithms that randomly access large amounts of data may have to be altered if such data is to be partitioned.

Ignoring such issues, the implementation of reliability tailoring is straightforward in TASK because most of the required information is already available. For example, the failure distributions of Cm*'s hardware components are constantly being collected by diagnostic software [29]. Furthermore, estimates of the failure rates of task force objects can be based on object size or complexity, or estimates can be provided by experienced application programmers. Alternatively, an existing software monitor [144] could be enhanced to gather reliability information.

3.7. Tailoring an Operating System

While we have used the PDE task force to demonstrate our tailoring methodology, it is important to note that programs of any size and complexity can be tailored. As an example, consider the most complex task force executing on Cm*: the STAROS operating system [80]. STAROS was described as a connected graph of functionally separable code modules. Each code module either corresponded to a process in the executing operating system or it corresponded to a collection of routines and associated data structures [78, 77]. The arcs in the graph were labelled by the access frequencies between code modules[15]. Therefore, the description of STAROS resembled a proximity blueprint as illustrated below:

[15] Access frequencies were derived from several benchmark application programs [28].

Objects: {the code modules of the operating system}

Relations: {proximity relations expressing the frequency of interaction between code modules}

STAROS was tailored to minimize the time required to service user requests. Multiple tailoring experiments were performed, resulting in multiple operating system versions which were identical in functionality and different in performance [77]. Two tailoring strategies were pursued. The first strategy was to trade space for time. Specifically, by replicating selected code modules, their execution times were reduced, thereby improving their service times.

The manner in which replication reduced execution and service times in outlined below. Consider an invocation of a code module that is made by a program executing on a processor not containing the module. In STAROS, this invocation resulted in the remote execution of the module, which generated large amounts of non-local memory references. Since non-local are slower than local memory references, replicating the code module improved its execution time, thereby reducing its service time. In addition, code replication reduced the likelihood of exceeding the Kmap's processing capacity since fewer non-local memory references were made.

The second tailoring strategy pursued was to alter the cost of accessing system services, thereby again improving service times. Initially, each system service was requested by execution of an expensive (slow) call operation (similar to UUOs in Tops-10 or *kernel calls* in Hydra [162]). Later, frequently used system services were requested via cheap (fast) message facilities, and several processors were dedicated to execute the processes servicing those requests. Although the performance of this version was not measured, a queueing model similar to Raskin's validated model [135, 77] indicated that system throughput would not degrade in large Cm* configurations.

3.8. The Proximity Model-Discussion and Extensions

In this section, the advantages, disadvantages, and alternatives to our use of the proximity model are discussed. We conclude the section by presenting an extension of this research.

One advantage of the proximity model is that it provides a framework in which a rich variety of multiple processor application programs, distributed hardware, and tailoring objectives can be expressed. In this chapter hardware instances were formulated for the Cm* architecture and for computer networks. Software instances were formulated for the PDE task force and the STAROS operating system. Because the PDE task force is executed as an application under the STAROS operating system, the types of objects in the PDE model instance are those that are supported by STAROS. The objects used in the model instance for STAROS were different; they corresponded to the logical units of the functional decompositions of the STAROS system. The tailoring objectives discussed were the improvement of speedup and the improvement of reliability. Reliability tailoring involved the definition of two additional model instances.

Another advantage of the proximity model is that it provides a flexible basis for the automation of tailoring. The TASK system demonstrates this. Moreover, TASK serves as a testbed for experimentation with varying tailoring objectives and metric functions. In addition, the TASK implementation of the proximity model permits different degrees of programmer involvement. Specifically, programmers can rely on TASK's built-in tailoring knowledge, or they can overrule TASK's automatic tailoring decisions, or they can formulate their own tailoring objectives or metric functions.

Furthermore, the proximity model is a basis for comparisons to related research. For example, the complexity of tailoring will be estimated by comparison to mapping problems investigated elsewhere. In addition, references to related research act to guide the development of the automatic tailoring procedures presented in Chapter 5.

There are several disadvantages to the manner in which the proximity model is used in TASK.

These disadvantages are caused by the omission of certain information from model instances, and they are due to the fact that TASK uses only a single hardware or software model instance per run. An example of the omission of information is the insufficient detail in the model instances concerning the algorithmic code. Specifically, model instances do not record sufficient detail about algorithmic code to indicate whether objects can safely be replicated or partitioned. As a result, programmers must perform replication and partitioning explicitly. Furthermore, *precedence relations* [58, 23] expressing the control flow in a task force are omitted from model instances, since such relations cannot be modeled by non-directional arcs. As a result, programmers must explicitly specify the parallelism in a task force. Since TASK only uses a single software and hardware instance per run, each of which is assumed constant, these instances do not accurately reflect the dynamically varying computation of a task force or the dynamic variation of the hardware configuration. For example, hardware contention is ignored, although such contention can alter the effective bandwidth of a bus.

Alternatives to the manner in which the proximity model can be used for tailoring can be classified into three categories: tailoring at different grains of description, tailoring at different times, and tailoring using different scopes. Consider the grains of description. In TASK, a task force is described in terms of its data and code objects and its processes. At this grain of description, automatic tailoring cannot result in requirements for changes to the algorithmic code. For example, if the logic of the code requires that an object be shared, then tailoring cannot cause that object to be replicated. Alternative grains of description range from very fine to very coarse grains. If we used the very fine grain of description at which the individual instructions and variables of the algorithmic code are specified, then tailoring could be extended to include the automatic replication of objects as well as the automatic generation of parallelism [89]. Since such powerful tailoring requires algorithmic code alterations, it is not possible at the next coarser grain of description, namely the grain at which procedure and parameter specifications appear in TASK programs. However, at this grain parameter and procedure types can be checked [156]. In addition, one among several alternative procedures could automatically be chosen for inclusion into the code [140, 11, 159]. For example, the automatic

choice among procedures implementing alternative process communication protocols could be implemented [122]. Furthermore, if procedures are described in conjunction with their associated data structures, then we can automatically combine those descriptions into processes that can execute in parallel [69]. A grain of description coarser than that of TASK was used to tailor the STAROS operating system (see Section 3.7).

The second category is the time at which tailoring is performed. In TASK, tailoring is performed after task force compilation and prior to loading. This simplifies the implementation of the TASK system. However, tailoring could well be performed at other times. For example, as part of related research efforts, the automatic selection among alternative procedures has been performed at compile-time [105, 140, 11, 159] and at load-time [122], object placement decisions have been made both at runtime and at link-time [38, 52, 76], and process scheduling has been performed both at load-time and at runtime [150, 10, 34, 128]. In addition, load-time procedures have been designed [18] and runtime procedures have been implemented [139] to distribute data files and the accesses to such files within distributed data bases.

The third category is the scope of tailoring. Consider that automatic tailoring decisions in TASK are made for one task force at a time. As a result, the scope of tailoring is limited. Although we could tailor several task forces at a time, the required computation time would be prohibitive. Currently, a large task force can be tailored in a few minutes of computation time. An enlarged scope of tailoring would increase this computation time because more information would have to be processed. Computation time can be reduced in two ways: by reducing scope or by using less detailed information. As an example of a reduction in scope, consider tailoring a single process in a task force by selectively *localizing* [149] the code and data objects of the process. As an example of the use of less detailed information while the scope is increased, we consider the scheduler designed for the Medusa system [128]. This scheduler attempted to "tailor" the entire system by assigning the processes of all currently executing task forces to processors on the basis of process execution priorities. The objective of tailoring was to maximize useful parallelism [129].

We conclude by discussing one means of extending the hardware model instance presented in Section 3.2. In this model instance, the physical parameters of distributed hardware are recorded. As additional parameters, information specific to the operating system could be included into the instance. For example, the instance could be extended to record the manner in which STAROS process schedulers [138] are set up. As a result, then tailoring could be improved by assigning small processes to timeshared processors and large processes to dedicated processors. Alternatively, or in addition, TASK could generate multiplexer setup commands prior the execution of each task force.

3.9. Survey of Related Work

The proximity model's instances capture several essential characteristics of multiple processor applications and architectures. Consequently, descriptions that are similar to these instances have been used in a variety of related research. Related research topics include file allocation in distributed data bases [18, 151, 32], task scheduling in multiple processor systems [150, 27], and the design of computer networks [112]. We will consider each of these topics in turn.

File allocation in distributed data bases concerns the replication of files and their placement into the nodes of a computer network [18, 25]. Files are accessed by processes whose assignments to network nodes are fixed. In a formulation in terms of the proximity model, the following objects are identified:

Software Objects: {files,processes}, where each file has a size and a replication factor

Hardware Objects: {network nodes}, where each node has a fixed amount of memory

Two kinds of proximity relations are defined between a process and a file. In one kind, the frequency of update accesses is expressed. Update accesses touch all copies of the file. In the other kind, the frequency of read accesses is expressed. Read accesses touch only the nearest file copy. The proximity relations among network nodes are straightforward. They are typically phrased in terms of the average number of network links traversed by file access requests between nodes. Tailoring consists of replicating files to the least extent necessary and placing the replicated files into network

nodes. The tailoring objective is to guarantee that neither update nor read requests will be too slow [32]. The commonly used metric is a linear combination of the total amounts of memory space and interprocessor communication.

Research results can be summarized as follows. Eswaran [47] established that the optimal allocation of a known number of file copies [25] is an NP-hard problem. Despite this result, most solutions to the file allocation problem used computationally expensive integer programming methods [118]. The same methods were used to solve generalized file allocation problems in which software and hardware design parameters were determined during file allocation. Generalizations included the determination of (1) the assignment of processes to processors [119], (2) network properties such as link bandwidths, link reliabilities, and node reliabilities [107], and (3) software properties such as reliability requirements or maximally allowable file access times [107]. Typically, these generalized file allocation problems were simplified by restricting network topologies to tree-connected [19, 53, 73] or to star networks [55]. As a result, computational tractability was improved, and more realistic solutions could be attained [55]. Some authors used heuristic procedures rather than optimal algorithms to further improve computational tractability [107]. Additional references concerning file allocation are provided in overview papers by Chu [27] and Morgan [118].

Resource allocation problems similar to file allocation problems were considered by Ramamoorthy [133] who investigated the placement of relations in distributed, relational data bases, by Chu [26] who analyzed the performance of directory systems in star and distributed networks, and by Kung [91] who investigated the manner in which parallel algorithms could be mapped to array computers. Furthermore, early investigations regarding file allocation dealt with the linear topologies of memory hierarchies. Researchers investigated designing [1] and optimally using [132, 22, 76, 38, 37] memory hierarchies as well as improving the locality of programs [52, 4]. Solutions were attained with queueing models [86, 5] and with linear programming methods.

The second related research topic is task scheduling in multiple processor systems. Task schedul-

ing as well as workload balancing concerns the assignment of processes to processors during system operation. Gylys [61] formulated a quadratic zero-one programming problem to determine process assignments that equably partitioned total workload across available processors. Specifically, total interprocessor traffic (total communication cost) was minimized, while the number of processes that could be assigned to each processor was constrained. Since it is difficult to solve quadratic zero-one programming problems optimally, heuristic solution procedures were used. Stone et. al. [150, 151, 10] used efficient transportation algorithms to assign multiple, communicating code modules to 2 processors. Again, total communication cost was minimized. Extended solution procedures dealt less efficiently with the "n" processor case and with dynamic process rescheduling [10]. Cullman [35] extended Stone's work by considering a processor's current load factor prior to assigning a process to the processor. Brantley developed simpler solutions for a specific class of multiple processor applications: signal processing applications [12]. Theoretical treatments of task scheduling are concerned with additional topics: (1) the direct minimization of the total completion time of a set of tasks [85], (2) the derivation of complexity results for task allocation algorithms [58], and (3) the influence of resource requirements or precedence relations among tasks on task scheduling [23, 100, 168].

Non-mathematical formulations of the task scheduling problem were first suggested by Jenny [69] and Jones and Schwans [81]. Both used relational expressions (*proximity relations*) to express inter-process communication. The purpose of Jenny's formulation was to partition a large number of program modules into a smaller number of independently schedulable processes, whereas Jones and Schwans [81] employ proximity relations to allocate resources to given processes and objects.

The third related research topic is the design of computer networks [160]. Here, the requirements of potential network users are employed to derive suitable node and link numbers, locations, bandwidths, and reliability values for the future network [56, 112, 95, 158]. We elide the formulation of proximity model instances that capture network design requirements. Instead, we note that the large number of components involved requires the use of heuristic solution procedures [18, 25].

4. Using the Proximity Model

In multiple processor systems of substantial size, resource allocation cannot be performed by explicitly allocating particular hardware resources to individual software components. First, the number of individual allocation decisions is typically too large. For example, even a small task force constructed for a small configuration of Cm* consists of 20 different software components to which resources of any one of 10 different computer modules must be allocated. Second, since the configuration of distributed hardware of substantial size can change frequently [141], software cannot be constructed unless programmers check whether allocated hardware components are available. Therefore, if resources are explicitly allocated, programmers must know the precise configuration of the hardware used for software execution. Programmers should not be forced to acquire such knowledge. Third, if hardware can change, it is inappropriate to restrict software by means of explicit allocations to execute on specific hardware configurations.

In TASK, explicit resource allocation is not necessary. Instead, programmers state high-level directives in TASK programs that concern the usage of resources by sets of task force components. These directives are processed by TASK and are used to determine appropriate resource allocations automatically.

This chapter is structured as follows. First, the notion of resource usage directives is discussed and examples of directives and of their use are presented. Next, the syntax of directives is described. Last, we discuss the manner in which directives are processed by the TASK compiler.

4.1. Proximity Directives

In Chapter 3, it was shown that resource allocation can be performed on the basis of proximity relations. Accordingly, TASK's *resource usage directives* (also called *proximity directives*) express proximity relations [2] between the components of the executing task force. Specifically, proximity

directives express the frequencies of access of processes to passive objects, the potential access relations between passive objects, and the constraint relations between active or between passive objects. For example, a proximity directive might indicate the frequency of access of a process to a set of task force components, and an additional directive might express that this process can execute in parallel with several other processes.

In addition to expressing proximity relations between task force objects, directives can also express the proximity of task force objects to certain hardware components. Such directives are typically used to ensure the allocation of special-purpose hardware components to appropriate task force objects. For example, a task force process acting as a network server can be called "close" to a processor to which a network link is attached.

Proximity directives can encode different values of proximity relations. For example, a programmer can use two proximity directives with different values to express that a process accesses two different sets of task force objects with different frequency. However, in TASK such proximity values are not stated in terms of access frequency values because programmers typically do not know the precise frequencies[16]. Instead, proximity values are stated in terms more familiar to programmers (see Section 4.2.1).

The semantics attached to different values of proximity directives are explained in the next section. Here, we use an example to illustrate that directives have two different, useful interpretations. Namely, they can be interpreted as tailoring preferences or as tailoring constraints. For example, if a programmer states the close proximity of two task force objects, then the programmer may either prefer or assume that these objects be allocated memory in the same computer module. By default, the TASK compiler interprets such a directive as a preference, so that it is possible that the two objects will not be placed into the same computer module. The intent of this interpretation is to allow the compiler to use the stated directive, knowledge of the TASK program's execution blueprint, and knowledge of the

[16] *An extension of our research is the automatic verification of proximity directives by an intelligent task force monitor.*

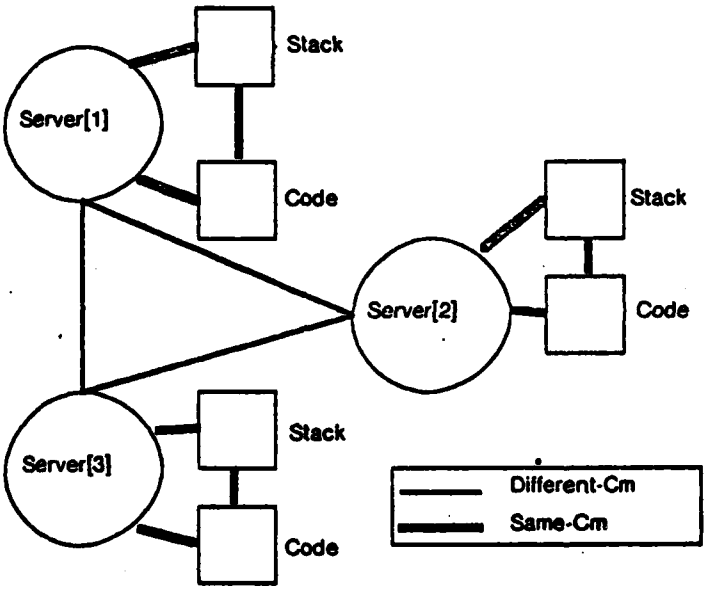


Figure 4-1: Some Directives Between Three Server Processes

current Cm* hardware configuration to make appropriate allocation decisions. However, if programmers require explicit control of resource allocation, the TASK compiler can be instructed to interpret proximity directives as constraints. In that case, the directive above implies that allocation decisions that do not place the two objects into the same computer module are illegal. We note that directives relating task force to hardware components are always interpreted as constraints because such directives are typically used to ensure the allocation of specific hardware to task force components.

Whether proximity directives are interpreted as preferences or as constraints, typical directives in TASK programs do not touch upon each one of the components of the specified task force. As a result, the tailoring decisions made by the TASK compiler will substantially influence the performance

of the executing task force. For example, consider the PDE task force for which the following directives expressing constraints are specified, as illustrated by figure 4-1:

- the coordinator process and each server process should be assigned to the computer modules into which the coordinator's or the servers' code and stack have been placed (these directives are represented by the thick lines in the figure);
- all server processes should be assigned to different computer modules (these directives are represented by the thin lines).

Several mappings of the PDE task force to the Cm* hardware satisfy the constraints above, and each mapping can result in different performance of the executing task force. Consider the following sample mapping: a single server process is assigned to each of the cluster's computer modules, each server's code and stack are placed into the memory of the computer module to which the associated process has been assigned, and all grid partitions are placed into the memory of a single computer module of the cluster. Furthermore, the coordinator process shares the use of one processor with one of the server processes. In Chapter 3, it was shown that the PDE task force mapped in this fashion exhibits linear speedup. Therefore, this mapping constitutes a good set of allocation decisions. However, given the directives above, another sample mapping is one in which the server processes and their code and stack objects are spread across two clusters while all grid data remains in a single cluster. The PDE task force mapped in this fashion does not exhibit linear speedup because the workload of server processes are imbalanced.

4.2. Proximity Directives in TASK Programs

Proximity directives in TASK programs are strictly *local*, namely, each directive relates components within a single TASK template. Therefore, directives cannot directly state *global* relations in which the components of multiple templates in a TASK program are related to each other. Instead, several global relations are automatically derived from the the local directives stated in a TASK program (see Section 4.3). We have two reasons for using local directives. First, global directives would breach the rules of scope that are enforced within TASK programs. Second, as shown later, the syntax required

to state local directives is simpler than the syntax required for global directives.

Since proximity directives are local, a separate set of directives is associated with each TASK program template. Directives appear in postludes to templates so that a template's directives can be changed separately from the template's *Construction Description*. For example, in each module description, the *Resource-Usage Directives* appear after the module's *Construction Description* and after the *Function Descriptions*. Furthermore, each *Function Description* contains its own *Resource-Usage Directives*:

⟨Module Description⟩ ::= **Module** *⟨Complex Template Name⟩* (*⟨Formal Parameters⟩*)# **is**
⟨Construction Description⟩
⟨Function Description⟩+
⟨Resource-Usage Directives⟩

⟨Function Description⟩ ::= **Function** *⟨Complex Template Name⟩* (*⟨Formal Parameters⟩*)# **is**
⟨Construction Description⟩
⟨Resource-Usage Directives⟩

Resource-Usage Directives consist of the keyword **Directives** followed by a list of proximity directives relating the objects instantiated within the preceding *Construction Description*. For example, the sample proximity directives in the **DoCoordinate** function below state a proximity relation between two prospective process components: **Stack** and **Code**. As shown, the directives refer to the template components by their *Comp Names* (Ellipses indicate missing text, and "--" precedes comments. Keywords are underlined):

```
Function DoCoordinate (. . .) is
Construct (
        Stack: . . .
        Code: . . .
        Data: . . .
        . . .
        )
Directives (
        SameCm (Stack, Code) --  a sample directive
        )
```

Each instantiation of a process from this function contains the three components listed in the *Construction Description*, where two components are related by the **SameCm** directive. As will be

discussed in the next subsection, **SameCm** means that **Stack** and **Code** should be placed as close to each other as possible, preferably into the same computer module.

4.2.1. Degrees of Proximity

As mentioned in Section 4.1, the values of TASK's proximity directives are not stated in terms of frequency of access values because programmers do not know those frequencies. Instead, a few different *degrees of proximity* are formulated in terms of hardware characteristics known to programmers, such that each degree expresses a different proximity value. As a result, proximity degrees differ depending on the distributed hardware used for task force execution. In Cm*, each one of six different degrees of proximity denotes one choice concerning the mapping of task force objects to the three-level hierarchy of component interconnections. The terms "should" and "must" are used in the itemized list that follows because each proximity directive can be interpreted either as a tailoring preference or as a tailoring constraint (as explained in the previous section):

- **SameCm** expresses that the related objects should/must be in the same computer module;
- **SameCluster** expresses that the related objects can be in the same computer module and should/must be in the same cluster;
- **DiffCm** expresses that the related objects should/must not be in the same computer module and should/must be in the same cluster;
- **NearCm** expresses that the related objects should/must not be in the same computer module and can be in the same cluster;
- **DiffCluster** expresses that the related objects should/must not be in the same cluster;
- **NoCare** expresses that the related objects can be anywhere.

The **Diff...** and **Same...** proximity degrees are straightforward expressions of whether objects should be in the same or in different computer modules or clusters. The **NearCm** degree expresses a "different" relationship with respect to computer modules and a **NoCare** relationship with respect to clusters. Note that the converse degree expressing a "different" relationship with respect to clusters and a **NoCare** relationship with respect to computer modules is superfluous in Cm* because the

assignment of objects to different clusters implies their assignment to different computer modules.

Proximity degrees are readily formulated for architectures other than Cm*. For example, in the EtherNet access to any remote host is equally costly due to the high cost of packetizing and de-packetizing messages [115]. In this case, two proximity degrees must be distinguished: `SameComputer` and `DiffComputer`. These proximity degrees must be refined for other computer networks. Specifically, different costs of packetizing and de-packetizing on different network nodes must be taken into account if a network consists of inhomogeneous computers. Furthermore, if communication protocols differ within a single network, the resulting transfer costs also differ. In both cases, proximity degrees can be stated as vectors of distances between network hosts [18]. Note that the proximity degrees within the ArpaNet would be stated in this fashion (see Chapter 3).

4.2.2. Expressing Proximity Relations with Proximity Directives

Several sample directives serve to clarify the exact relationships between the proximity directives in TASK programs and the corresponding proximity relations in the task force being specified. Consider a server module in the PDE task force that contains three server processes: `Server1`, `Server2`, and `Server3`. The following directive in the *Resource-Usage Directives* of the server module expresses that each server process should/must execute on its own computer module in one Cm* cluster:

```
DiffCm (Server1, Server2, Server3)
```

This proximity directive conveniently states the proximity relations between the three server processes. Specifically, the directive expresses three different binary *constraint* relations (see Section 3.2) between the three active objects (server processes). If each of these relations were expressed separately, the following, equivalent, binary directives would have to be stated:

```
DiffCm (Server1, Server2)
DiffCm (Server2, Server3)
DiffCm (Server1, Server3)
```

In the directive above, three active objects (processes) are related. If a directive relates three passive objects, then the directive expresses a *potential access frequency* relation (see Section 3.2).

For example, the three server process components called **Code**, **Stack**, and **Data** are related as **SameCm** by the following directive in the *Resource-Usage Directives* of the *DoServe* function:

```
SameCm (Code, Stack, Data)
```

This directive is intended to express that each server process will access its code, stack, and data with the such frequency that the objects should have the proximity indicated by **SameCm** to the process. In TASK, a specific range of potential access frequencies corresponds to the proximity degree **SameCm**, and these ranges are not visible to programmers. Each proximity degree denotes a different range of potential access frequencies. These ranges have been determined in tailoring experiments with the TASK system. Their precise delineation is elided here.

The sample directives above are straightforward expressions of either potential access frequencies between passive objects or constraint relations between active objects. The expression of the *frequency of access* of a process to its components is not as straightforward in TASK because it is not possible to state a global directive that relates a component of one template, the template containing the process, to components of another template, the function template of the process.

In Chapter 3, it was shown that frequency of access relations are important for task force tailoring. Consequently, their expression by means of directives must be possible. The approach taken in TASK is simple: the TASK compiler derives global proximity relations from the local directives specified. For example, the local directive above expressing the potential access frequencies between prospective server process components is used to derive a global relation expressing the frequencies of access of **Server1** to its components. As a result, the following (illegal) global proximity directive is derived from the specified local directive:

```
SameCm (Server1, Server1.Code,
                 Server1.Stack, Server1.Data)
```

The illegal directive states that the **Code**, **Stack**, and **Data** objects of the process **Server1** should be placed where the process is assigned. Note that pathnames are required to distinguish the components of **Server1** instantiated from the **DoServe** template from the components of **Server2** instantiated from the same template. Since frequency of access relations are derived and not stated,

the use of pathnames is avoided in TASK, thereby simplifying the syntax. Additional information concerning the derivation of global relations from local directives is provided in Section 4.3.

4.2.3. Explicit Allocations

The sample proximity directives in the previous section do not require that programmers know the configuration of the distributed hardware used for task force execution. However, some task forces require that specialized hardware resources be allocated to particular task force components. To accommodate such needs, the explicit allocation of hardware components to task force objects is supported in TASK. Specifically, programmers can first select a particular hardware component and then relate the selected component to a particular task force object. However, programmers need not know the physical names of hardware components. Instead, they can select hardware components by attribute, by functional, or by their virtual TASK-defined names.

Virtual hardware component names can be used in selections if the selected, virtual component is not assumed to exhibit explicitly desired attributes, such as memory of sufficient size or an attached disk. Hardware component names in TASK resemble array names in algorithmic languages. For example, the name of the second computer module in the first cluster of Cm* is Cluster1[2]. The same cluster is named by CmStar[1] or by `Cluster1`[17]. As with all virtual hardware component names, the TASK loader binds the names `Cluster1` and `CmStar[1]` to a specific physical cluster.

Once a programmer has selected a specific hardware component, this component can be related to a task force object by means of a proximity directive. The proximity degree `Same` is used in such directives. For example, the following directive `Same` expresses that the process `Server2` must execute on the second computer module in `Cluster1`:

```
Same (Server2, Cluster1[2])
```

[17] The Cm* architecture can be described by a few named arrays (see Appendix 1 for a complete list of predefined Cm* component names). However, architectures that are less regular in topology, such as the ArpaNet, require different naming schemes. In such cases, we expect that pathnames will be used.

A hardware component is typically selected once and then used in multiple directives of degree **Same**. To facilitate multiple uses of a selected component, variables can be defined and assigned as values the names of selected components. For example, the directive above can also be stated as:

```
MyCm: Cluster1[2]
Same (Server2, MyCm)
```

Since selections by name restrict the choices of the TASK system in resource allocation, we instead encourage programmers to select hardware components by functional, if possible. Specifically, the functional **AnyOf** can be used to express that a particular object can be assigned to any of a set of hardware components. For example, any one of the computer modules in the first cluster of **CmStar** is selected by:

```
AnyOf (CmStar[1])
```

This selection may but need not result in choosing the previously selected computer module **Cluster1[2]**.

The functional ***NumberOf*** *(not implemented) returns an integer count of the number of hardware components in the set to which it refers. For example, the following statement returns an integer count of the number of computer modules in* **Cluster1**:

```
NumberOf (Cluster1)
```

Given this functional, object instantiation can be made dependent on the availability of hardware resources (see Section 4.3.4).

Neither selections by name nor selections by functional are adequate if task forces require hardware exhibiting certain attributes. For example, a task force process acting as a terminal handler must be tested on a computer module to which a terminal is attached. If such a computer module is selected by name, programmers must know the mapping of virtual to physical computer module names as well as the physical computer module's attributes in order to guarantee that a terminal is attached to the selected module. Since the configurations of Cm* can change, a programmer cannot be expected to have this knowledge. Therefore, selections like the above require that hardware be

selected by attribute. The following selection in a TASK program will result in the selection of a computer module to which a terminal is attached:

```
TerminalModule: AnyOf CmStar[1] where (HasTerminal = True)
```

This computer module is in the first cluster of Cm*.

More than one attribute may be specified in a selection, and selections can be nested. For example, the first of the following selections expresses that a cluster containing (1) a large amount of available memory and (2) at least one computer module with an attached terminal should be chosen. The second selection chooses a computer module (in the selected cluster) to which a terminal is attached and which has more than 64K bytes of memory:

```
LargeCluster: AnyOf CmStar where
                (MpSize >= 500K and NumTerminals >= 1)
LargeTerminalModule: AnyOf LargeCluster where
                       (MpSize >= 64K and HasTerminal = True)
```

The TASK compiler reports selection failures if the devices or hardware components selected in a TASK program are not available in the current configuration of Cm*. Selection defaults are generated in such cases.

4.2.4. Language Summary

Three principles guide the design of TASK proximity directives. First, proximity directives are specified and changed separately from task force component declarations. Hence, directives are stated in postludes to templates. Second, directives are formulated in terms familiar to programmers. Consequently, proximity degrees do not directly express the frequency of access relation values in proximity blueprints. Instead, proximity degrees are formulated in terms of the hierarchy of memory access exhibited by the distributed hardware. Third, proximity directives are designed so that programmers state as little hardware detail as possible. As a result, component selections by functional or by attribute are preferable to selections by name. Additional detail concerning proximity directives is presented in Appendix 1.

4.3. Processing Proximity Directives

In this section, we discuss the derivation of proximity relations in an executing task force from the proximity directives in the TASK program describing the task force. Three kinds of relations are derived: constraint relations, potential access frequency relations, and frequency of access relations. As shown in the previous section, the constraint relations between active objects are readily derived from proximity directives. An n-ary directive relating the active objects merely has to be translated to equivalent binary relations. The translation of directives between passive objects to potential access relations between those objects is equally straightforward. Namely, each degree of proximity represents a specific range of potential access frequencies when passive objects are related to each other. However, it is not as straightforward to determine the global proximity relations between a process and its components from TASK's local directives. The required derivation of global from local directives is discussed next.

In TASK, the following general rule is applied to derive global from local directives. If one object is the *component of* another, then the types of the component and of its "parent" as well as the local directives that involve the component determine the proximity relation between parent and component. For example, consider a process instantiated from the `DoServe` function, where the directive **SameCm** `(Code, Stack)` is stated in the postlude of the function template. In this case, the *basic* objects containing the code and the stack are *components of* an object of type *process*. Furthermore, a directive with degree **SameCm** refers to the code and stack objects. Given these object types and this directive, the TASK compiler derives that the code and the stack objects are each related as **SameCm** to the process. Therefore, the proximity relation expressed by the (illegal) global directive **SameCm**`(Server1, Server1.Code, Server1.Stack)` is derived from the local directive **SameCm** `(Code, Stack)`.

Frequency of access relations that involve other than *basic* components of *processes* are determined in an analogous fashion. We do not discuss these and other automatic derivations in detail. In

addition, we elide discussion of the defaults generated by the compiler concerning objects not related by directives. However, we note that appropriate automatic derivations and defaults have been determined by extensive experimentation[18].

4.3.1. Transitive and Conflicting Directives

Proximity directives are transitive with respect to "same" relationships. An example of a transitive directive concerning the "same" computer modules is:

```
SameCm (Code, Stack)
SameCm (Stack, Data)
```

These directives imply that the following additional directive is redundant:

```
SameCm (Code, Data)
```

An example of a transitive directive concerning the "same" clusters is:

```
SameCluster (Code, Stack)
SameCluster (Stack, Data)
```

These directives imply that the following additional directive is redundant:

```
SameCluster (Code, Data)
```

Directives are not transitive with respect to "different" relationships. For example, the following directives imply nothing about the relation between `Server1` and `Server3`:

```
DiffCluster (Server1, Server2)
DiffCluster (Server2, Server3)
```

Since programmers may state directives that exhibit transitivities or that conflict with each other, the TASK compiler must detect and resolve transitivities and conflicts. For example, consider the binary directives below:

```
SameCm (Code, Stack)
SameCm (Stack, Data)
```

These directives are combined into one ternary directive:

```
SameCm (Code, Stack, Data)
```

---

[18] Several sample task forces and prior experimentation with Cm* [29, 28] were used to determine appropriate defaults and automatic derivations [154].

However, the following directives are not simplified in this fashion:

```
SameCm (Code, Stack)
SameCluster (Stack, Data)
```

Instead, the only transitivity effect recorded is that `Code`, `Stack`, and `Data` should/must be placed into the same cluster.

Conflicts are caused both by the explicit statement of conflicting directives and by the effects of transitivity. For example, the following direct conflicts between directives will be discovered:

```
SameCm (Code, Stack)
DiffCm (Code, Stack)
```

In addition, the compiler will also discover the conflicts due to transitivity in the directives that follow:

```
SameCm (Code, Stack)
SameCm (Stack, Data)
DiffCm (Code, Data)
```

Since transitivities and conflicts among directives are determined on a template by template basis, their number is typically small. As a result, transitivity and conflict detection procedures employ straightforward algorithms and data structures. As data structures, we employ proximity sets, where the *proximity set* of a directive is defined as the set of task force components listed in the directive. Each proximity set is recorded by two bitvectors. One vector records "same" or "different" relationships with respect to computer modules, the other vector records "same" or "different" relationships with respect to clusters. Each bitvector records the objects that are set members by their positions within the template. Since a conflict or a transitivity between two directives cannot occur unless the associated proximity sets intersect, the detection of conflicts and transitivities includes the calculation of proximity set intersections. These intersections are determined by *And*-ing the involved bitvectors.

In the example of two transitive directives (see page 107), the compiler detects that the component at position `Stack` is a member of two proximity sets, each of which records "same" relationships with respect to computer modules and clusters. Both proximity sets are merged, thereby combining the directives. Merging consists of *Or*-ing the bitvectors that represent the proximity sets. However, in

the example that follows, the `Stack` component is only required to be in the same cluster as the other two components. Therefore, the involved proximity sets cannot be merged. However, the bitvectors expressing "same-ness" with respect to clusters are *Or*-ed. As demonstrated by these examples, in general, transitivity detection and correction consists of forming the transitive closures of those proximity sets in a TASK program whose intersections are non-empty and whose proximity degrees are identical with respect to "same" relationships concerning clusters or computer modules.

Conflicts are determined much like transitivities. For example, a conflict concerning computer modules (see page 108) is detected by *And*-ing the directives' computer module bitvectors. If one bitvector specifies a "same" relationship while the other specifies a "different" relationship, then an *And* operation resulting in a non-empty bitvector signifies a conflict. Conflicts are resolved by use of a ranking among proximity degrees. Higher ranked degrees take precedence over lower ranked degrees. While we do not explain the rankings in detail, we note that programmers are always notified of detected and corrected conflicts.

Because the TASK compiler resolves transitivities and conflicts, they need not be considered by the tailoring procedures in Chapter 5. However, such procedures must take into account that conflicts can occur during resource allocation. As an example of a conflict during allocation, consider the directives:

```
SameCm (a,b); NearCm (b,c); DiffCluster (a,c)
```

When tailoring procedures choose resources for `a`, `b`, and `c`, a conflict arises if the three objects are allocated to the same cluster. This conflict cannot be determined before resource allocation because the `NearCm` directive permits the allocation of objects `b` and `c` to the same or to different clusters. Tailoring procedures check and correct allocation conflicts in an optional second phase after resource allocation has been decided (see Chapter 5). Note that this phase need not be run when directives are interpreted as preferences.

4.3.2. Directives and Selections

While most proximity directives express proximity relations between task force components, directives with the degree **Same** determine the hardware components that can be used as resources for particular task force components. The set of hardware components that can be allocated to an object is called the *eligibility set* of the object. Given the assumption that each object can initially be assigned to any hardware component, statement of a directive with the degree **Same** is equivalent to a restriction of the eligibility set of the object named. Therefore, when processing directives with the degree **Same**, the TASK compiler restricts the eligibility sets of task force components. Given the proximity directives presented, the following situations have to be dealt with during such restrictions.

- *Multiple selections*. If a single object is involved multiple directives with the degree **Same**, this object is directly dependent upon multiple selections. In this case, the TASK compiler must compute multiple restrictions of the object's eligibility set. The selections and directives below provide an example of multiple selections applied to one object:

```
DiskCm: AnyOf Cluster1 where (HasDisk = True)
EtherCm: AnyOf Cluster1 where (HasEther = True)
Same (EtherCm, Coordinator)
Same (DiskCm, Coordinator)
```

- *Successive selections*. If a selection is itself derived from a restricted set of hardware components, then the TASK compiler must successively restrict the eligibility sets of the associated task force objects. The selection of **DiskCm** is an example of a selection from the restricted set **DiskCluster**:

```
DiskCluster: AnyOf CmStar where NumDisks >= 1
DiskCm: AnyOf DiskCluster where HasDisk = True
```

- *Empty selections*. Multiple and single selections can result in empty eligibility sets of the associated task force objects. The example of selecting a network link in Cm* demonstrates this point. There exists no computer module to which both an EtherNet link and a direct access link (**DALink**) are connected. Therefore, the selection below results in a empty set:

```
EtherDaLinks: AnyOf Cluster1 where
                  (HasEther = True and HasDALink = True)
```

As a result, the following directive relates the coordinator process to an empty selection:

```
Same (EtherDALinks, Coordinator)
```

To avoid empty selections, the TASK compiler ignores the last selection statement or the last attribute stated within a selection that caused the set of hardware components to become empty.

- *Inherited selections*. When computing the transitive closure of proximity sets, the eligibility set of any object within the closure is defined as the intersection of all eligibility sets attached to members of the closure. The following directive provides an example in which object b inherits the restriction attached to object a. Both objects must be placed into a computer module with a disk:

```
Same (DiskCm,a)
SameCm (a,b)
```

  To process inherited selections, the TASK compiler must first compute the transitive closures of proximity sets and then form the closures of all eligibility sets of the elements of the closures.

- *Conflicting selections*. Multiple directives with the degree Same in conjunction with other directives may lead to conflicts. For example, the directives that follow must lead to a conflict if there is only one computer module to which a disk is attached or if there is only one disk per cluster in Cm* (recollect that DiffCm also expresses that computer modules must be in the same cluster):

```
Same (DiskCm,a)
Same (DiskCm,b)
DiffCm (a,b)
```

  This conflict can be determined prior to resource allocation. However, if multiple computer modules with attached disks exist in a single cluster, then tailoring procedures must act to avoid conflicts during resource allocation. These conflicts can be avoided only if all proximity sets and all associated eligibility sets of which each task force component is a member are taken into account by tailoring procedures.

There are more examples of difficulties caused by the interplay between explicit allocations and other directives. Instead of dealing with the full generality of this interplay, in TASK the semantics of directives are simplified by imposing the following restrictions. First, for each object, exactly one directive with the degree Same can be stated. Consequently, multiple selections cannot occur. Second, objects can either be related to a hardware component or to other task force objects, but not to both. As as result, conflicting and inherited selections are avoided. Given these restrictions, the transitivities and actual conflicts between directives can be determined prior to tailoring. However, it is an inconvenience that users cannot state both a directive with the degree Same and with directives other degrees concerning a single task force object. Specifically, it is not possible to state directives that single out one member of a proximity set for "special treatment". For example, the following statements are illegal:

```
DiffCm (Process1, Process2, Process3)
Same (DiskCm, Process3)
```

Since these directives are illegal, there is no convenieht way to assign **Process3** to a specific computer module and to restrict the process to execute on computer modules different from those allocated to the processes **Process1** and **Process2**. The alternative directives below do not have the desired effect because either of **Process1** or **Process2** could also be assigned to the selected **DiskCm**:

```
DiffCm (Process1 Process2)
Same (DiskCm, Process3)
```

In cases like these, TASK users must explicitly assign each process to a specific computer modules.

We return to discuss a design choice taken at the beginning of this subsection, namely, to perform eligibility set restrictions while a TASK program is being parsed. The major benefit of restriction processing at parse-time is that directives can be pre-processed by the TASK compiler prior to their use by tailoring procedures. This was not the case in the original design of the TASK language in which the semantics of the **AnyOf** functional required that selections be performed *during tailoring*. Since **AnyOf** selections can be nested, chains of such selections had to be recorded. These chains had to be preserved until tailoring procedures chose resources compatible with the selections in the chains. Each time a choice was made, an entire chain had to be followed to ascertain that the chosen resource was a member of each listed eligibility set. Since this imposed considerable complexity on tailoring procedures, we discounted this interpretation of the **AnyOf** construct.

We note that eligibility sets are implemented like proximity sets; space-efficient bitvectors encode these sets and are used to compute set restrictions and to check for empty selections.

4.3.3. Using Selections to Parameterize Task Force Construction

Recollect that the server processes in the PDE task force must be assigned to dedicated processors in order to achieve desired speedups (see Section 3.5). Consequently, if the PDE task force is to

execute efficiently within a single cluster of Cm*, programmers should not instantiate more processes than there are available processors. Therefore, server replication should depend on the number of processors available.

Server replication dependent on the number of available processors is a typical example of the use of configuration knowledge by programmers. TASK's hardware component selections can reduce the amount of configuration knowledge required of programmers. Specifically, the functional **NumberOf** can be used to declare variables that express the number of resources available. For example, the following statements first select one large cluster of Cm* and then define the variable **CmCount** that encodes the number of processors in this cluster:

```
MyCluster: AnyOf CmStar where (NumCms >= 8 and NumDisks >= 1)
CmCount: NumberOf MyCluster
```

TASK syntax permits the use of the variable **CmCount** to instantiate a variable number of server processes (not implemented):

```
(i=1..CmCount) Server[i]: New Process ...
```

The example above is only one instance of the use of configuration knowledge during task force construction. Other instances are the partitioning of data in accordance with the number of available memory modules and the replication of objects in accordance with hardware component attributes. For example, server processes could be replicated according to the number of processors with local memory sufficient to place the servers' code and data. Furthermore, conditional construction is possible. For instance, certain objects may be instantiated only if certain devices are available[19].

---

[19] Languages designed for real-time systems have long recognized the importance of including hardware attributes into software specifications [17, 157].

4.4. Using the Proximity Model-Discussion and Extensions

By use of proximity directives, programmers interact with TASK to guide resource allocation. Specifically, programmers provide information to TASK that cannot be deduced from TASK or Bliss programs. For example, proximity directives stating the processes that can execute in parallel cannot be automatically deduced by compile-time analysis of control flow in algorithmic code [89].

There are additional ways in which programmers can interact with the TASK system. For example, programmers can experiment with different tailoring objectives, metric functions, and tailoring procedures. However, to facilitate such experimentation, the current implementation of TASK has to be augmented by a monitor of task force execution. Such a monitor could be used to verify that proximity directives accurately reflect access frequencies observed during task force execution. Furthermore, the observed access frequencies could be displayed to programmers and could be used to update the proximity directives stated in the TASK program. In addition, this monitor could collect information that cannot be determined statically. An example of such information is the actual availability of hardware resources, such as the load on selected processors.

We have completed a preliminary design of an augmentation of TASK by monitoring facilities. This design is based on ascribing unique names to task force and hardware components. These names are jointly used by TASK and by a task force monitor. Two unexpected conclusions could be drawn from the preliminary design. First, the relations and attributes described in execution and proximity blueprints can be used to filter [144] the potentially overwhelming amounts of information collected at task force execution time. Second, we expect that tailoring procedures using information about both the specification of a task force and its execution characteristics will perform better than procedures based on either one.

We have shown that the proximity model presented in Chapter 3 and the proximity directives presented in this chapter can be used to perform task force tailoring for computer networks. In this context, an interesting characteristic of proximity directives is that we distinguish between the inter-

pretations of proximity directives as tailoring constraints or as tailoring preferences. This distinction is important because constraints cannot be fulfilled without correct information, which is hard to obtain in distributed systems, whereas improvements in resource usage merely may not be realized if information is incorrect.

5. Tailoring-Modeling, Heuristics, and Experiments

The subject of this chapter is the automation of tailoring. Automation is based on the proximity model. Specifically, both the hardware and software model instances presented in Chapter 3 and the speedup tailoring metric function are formulated mathematically. These formulations are the basis for the development of automatic tailoring procedures. Specifically, we show that tailoring can be performed by state of the art optimization procedures. However, since mapping problems related to the problem of mapping a task force to Cm* have been shown NP-complete, heuristic rather than optimal automatic tailoring procedures are developed for the TASK system. These heuristics make use of standard techniques to arrive at tailoring decisions. Consequently, the quality of the heuristics' tailoring decisions is not evaluated. Instead, several typical tailoring examples are presented to illuminate the behavior of the heuristics. In addition, the amounts of space and time required to execute the heuristics are discussed.

5.1. A Mathematical Formulation of Speedup Tailoring

In this section, mathematical formulations of the software and hardware instances of the proximity model precede the presentation of a sample instance of the PDE task force. Given this sample instance, we show that the information contained in the mathematical formulation of the software instance can be derived from from a TASK program. Next, the mathematical expressions for the metric functions of the speedup tailoring objective are formulated.

5.1.1. Software and Hardware Instances

In the software instance of the proximity model, passive objects are distinguished from active objects. Passive objects contain code or data, and they are used for process communication. Passive objects are *placed* into memory. Active objects are processes, and they are *assigned* to processors for execution. Formulated mathematically, the active and passive objects are represented by two

sets. The set K^* contains the processes instantiated in a TASK program:

$K^* = \{0, \ldots, k \ldots m, \ldots K\} \quad k,m \in [0,K]$

The set I^* contains the passive objects:

$I^* = \{0, \ldots, i \ldots j, \ldots I\} \quad i,j \in [0,I]$

The intersection of I^* and K^* is the empty set. The *size* of the ith passive object (δ_i) is defined as the number of words of memory space required to store the object. For simplicity, it is assumed that the runtime representation of each process is placed wherever the process is assigned.

Recall that there are three kinds of proximity relations between active and passive objects. These relations are represented by the following sets of variables:

c_{ij} · the potential access frequency relation value between passive objects i and j
$c_{ij} \neq 0 \ \forall i,j \in I^* \quad c_{ii}$ = some default value $\forall i \in I^*$

d_{ik} · the frequency of access relation value between passive object i and process k
$i \in I^* \ k \in K^* \ d_{ik} \neq 0 \ \forall i,k$

w_{km} · the tailoring constraint relation value between processes k, $m \in K^*$
W_{kk} = some default value $\forall k \in K^* \ w_{km} \neq 0 \ \forall k,m \in K^*$

The hardware instance of the proximity model contains information concerning the distributed architecture. In the mathematical formulation of this instance, two sets are defined. The set P^* contains the available computer modules:

$P^* = \{0, \ldots, p \ldots q, \ldots P\} \quad p,q \in [0,P]$

For each computer module p, the variable σ_p records the amount of available memory (in number of words). The clusters in Cm* are contained in the set R^*:

$R^* = \{0, \ldots, r \ldots s, \ldots R\} \quad r,s \in [0,R]$

Given the sets of computer modules and clusters, the variables v_{pq} are defined to represent the proximity values between the processors of computer modules p and the memory units of computer modules q. In Cm*, three different proximity values exist:

v_{pp} - the proximity of a processor to the memory unit in the same computer module

v_{pq} - the proximity of a processor to a memory unit in the same cluster
$p,q \in r, p \neq q$

v_{pq} - the proximity of a processor to a memory unit in a different cluster
$p \in r, q \in s, r \neq s$

The general assumption stated in Chapter 3 is that any object can be assigned or placed anywhere in Cm*. However, in Chapter 4, eligibility sets are introduced so that programmers can restrict the hardware components to which software objects can be assigned. To express eligibility sets mathematically, two additional sets of variables are defined:

N_i - the set of indices of computer modules into which passive object i can be placed

Q_k - the set of indices of computer modules to which process k can be assigned

Tailoring is defined as performing a mapping of a task force's software instance to a hardware instance of Cm*. Since the software instance and the hardware instance each contain two kinds of objects, passive and active objects in the software instance and processors and memory units in the hardware instance, two sets of *decision variables* are introduced:

$$X_{ik} = \begin{cases} 1 & \text{if passive object i is associated with (defined below) process k} \\ 0 & \text{otherwise} \end{cases}$$

$$Y_{kp} = \begin{cases} 1 & \text{if process k is assigned to processor p} \\ 0 & \text{otherwise} \end{cases}$$

Proximity relation values are derived from proximity directives in TASK programs. These derivations have already been explained in Chapter 4. However, to clarify the manner in which the derived relation values are represented in the sets defined above, we provide an example. Consider an excerpt of the PDE task force's proximity blueprint. This excerpt contains two server processes each of which has one code and one stack component. Proximity directives state that servers should execute on *different-cms* (the *DiffCm* directive) and that the servers' code and stack should be

same-cm. In the mathematical formulation, this situation is represented as follows (for simplicity, passive and active objects are numbered consecutively.). First, the sets I^* and K^* are defined. I^* contains four objects, two objects per server process. These objects are numbered from 0 to 3:

I^* = {Code (0), Stack (1), Code (2), Stack (3)}

K^* contains the two server processes, numbered 0 and 1:

K^* = {Server(0), Server (1)}

The proximity relation values between the passive objects are directly derived from the stated directives. They are shown in the following table ("def" stands for "default value", which is a value automatically inserted by the TASK compiler):

	Code (0)	Stack (1)	Code (2)	Stack (3)
Code (0)	*def*	*SameCm*	*def*	*def*
Stack (1)	*SameCm*	*def*	*def*	*def*
Code (2)	*def*	*def*	*def*	*SameCm*
Stack (3)	*def*	*def*	*SameCm*	*def*

Similarly, the proximity relation values between processes are the following:

	Server (0)	Server (1)
Server (0)	*def*	*DiffCm*
Server (1)	*DiffCm*	*def*

Recall that the proximity relation values between processes and their components are derived by the TASK compiler. In this derivation, the compiler employs the *component of* relations to determine the components of each process. In addition, the compiler employs the *types* of process components and the proximity directives that state potential access frequency relations between process components. In this example, the following relation values are derived:

	Server (0)	Server (1)
Code (0)	*SameCm*	*def*
Stack (1)	*SameCm*	*def*
Code (2)	*def*	*SameCm*
Stack (3)	*def*	*SameCm*.

5.1.2. The Metric Function

Having expressed the available tailoring information, we now formulate the metric function for the speedup tailoring objective. Recall that this function expresses the total amount of communication in the executing task force. To formulate the function mathematically, two assumptions are made. The first assumption concerns the values of the relations d_{ik}, c_{ij}, and w_{km}. It is assumed that these values are expressed in the same unit, where the particular unit chosen is irrelevant to this discussion (e.g. words/sec or bits/month). To achieve this uniformity, an unusual interpretation of the constraint relations w_{km} is used. Specifically, the values of w_{km} are interpreted as measures of the amount of communication between the related processes. High amounts of communication suggest the assignment of the processes to the same processor, whereas smaller amounts suggest that the processes can be assigned to different processors within the same or within different clusters.

The second assumption concerning the formulation of the metric function relates to the manner in which tailoring is performed. Specifically, since passive object placement decisions and process assignment decisions cannot be made independently of each other (see Chapter 3), it is assumed that passive objects are first associated with certain processes and are then placed into memory in conjunction with process assignment. The passive objects associated with a process are placed into the memory of the computer module to which the process is assigned. As a result of this assumption, the metric function will be formulated in two parts. In one part, values are ascribed to the associations of passive objects with processes. In the other part, values are ascribed to the assignments of processes to processors.

Passive Object Association. The part of the metric function that ascribes a value to the association of passive object i with process k is a linear combination of several terms, called PartA, PartB, and PartC. The innermost term of the linear combination, PartC, represents the total value of the ith passive object's relation to the passive objects j that have already been associated with processes m. In other words, the strength of the ith passive object's relation to the passive objects of processes m is computed. PartC is computed as the sum of the appropriate relation values c_{ij}:

$$\text{Part C:} \quad \sum_{j \in I^*} c_{ij} X_{jm} \qquad m \in K^* \quad i \in I^* \tag{1}$$

In the next term, PartB, the sum of the relation values between process k and processes m is computed, where each relation value w_{km} is multiplied by the value computed in PartC. (The symbol "*" is used both as a superscript and as a multiplication symbol.):

$$\text{Part B:} \quad \sum_{m \in K^*} w_{km} * \text{Part C} \qquad k \in K^* \tag{2}$$

The complete metric function for passive object association is represented by PartA. In this term, the value of the access frequency of process k to passive object i is multiplied by the value computed in PartB:

$$\text{Part A:} \quad X_{ik} d_{ik} * \text{Part B} \tag{3}$$

This term expresses the total value of the existing proximity relation values concerning passive object i and process k. We note that the values that are ascribed to proximity relations are such that minimal communication in the executing task force corresponds to maximal values of the metric function.

Combined, this metric function appears as follows:

$$X_{ik} d_{ik} \sum_{m \in K^*} w_{km} \sum_{j \in I^*} c_{ij} X_{jm} \tag{4}$$

The weights and the scaling functions that are attached to the values of d_{ik}, w_{km}, and c_{ij} are not shown here[20]. However, we note that appropriate weights and scaling functions are chosen in conjunction with the particular units used for proximity relation values [154].

---

[20] The weights and scaling functions used in this research were derived as part of experiments with several task force examples.

Several characteristic of this formulation should be noted. First, the form of this metric function shows that the passive object association problem is an instance of a zero-one quadratic integer programming problem [61]. Object association is a zero-one integer programming problem because the decision variables can only have the values zero or one. The object association problem is called quadratic because the decision variable X appears twice in the metric function. This means that the decisions concerning the association of passive object i to process k is dependent on the decisions concerning the associations of objects j to the processes m. We note that this association problem has been shown NP-complete [47, 61].

To reiterate, the metric function in equation 4 does not express the cost of placing passive objects into memory. Instead, it computes a value that corresponds to the total amount of communication resulting from the association of passive objects with processes. The assignment of processes to processors is evaluated by a different metric function. As a result, two steps are required to map a task force to Cm*. First, passive objects are associated with processes, and then processes are assigned to processors.

The use of two separate mapping steps reduces the complexity of mapping a task force because the number of different mappings is reduced. Specifically, instead of testing all possible associations of each passive object with processes while each time also testing all possible assignments of processes to processors, the possible passive object associations and process assignments are tested individually. For example, consider a small PDE task force consisting of 5 processes, 4 component objects per process, and 10 grid partitions. Assume that the configuration of Cm* used for tailoring contains 20 computer modules. If placement and assignment are performed in one step, then the following number of possible passive object placements and process assignments must be tested:

((5 * 4 + 10) * 20) * (5 * 20)

However, if placement and assignment are performed in two steps, a smaller number of choices must be tested in total:

((5 * 4 + 10) * 20) + (5 * 20)

The technique that has been used to reduce the size of the task force mapping problem is an instance of a standard technique, called component *clustering* [112]. In this case, passive objects are clustered around processes before they are placed into memory. Note that clustering techniques similar to the one used here are commonly used to reduce the size of other large problems, one of which is the design of computer networks [112].

Since process assignment and passive object placement must be performed within the constraints of available memory, we define the *size* of a process (μ_k) as the sum of the sizes of the passive objects that are associated with the process. Note that passive objects are always associated with exactly one process, regardless of whether other processes can access them. As a result, each passive object's size is accounted for exactly once.

Process Assignment. So far, only the metric function for passive object association has been developed. In the following, we present the metric function used for the assignment of process k to processor p. Three terms are distinguished within this metric function. The first term computes the fraction of total communication due to the communication of process k assigned to processor p with processes m assigned to the same processor p. The value of this fraction is determined by the amount of communication between process k and processes m (w_{km}) as well as the cost of communication via local memory (v_{pp})[21] (Y^T_{mp} stands for the 'transpose' of the decision variable Y_{mp}, reversing the indices m and p. This is required to permit the multiplication of Y_{kp} by Y_{mp}.):

$$v_{pp} \sum_{\substack{m \in K^* \\ m \neq k}} Y_{kp}\, Y^T_{mp}\, w_{km} \qquad p \in P^* \quad k \in K^* \tag{5}$$

The second term computes the fraction of total communication due to the communication of process k assigned to processor p with processes m assigned to processors q in the same cluster:

---

[21] Note that we *are assuming* that the object used for communication between processes is a local buffer object. This assumption is not entirely true in Cm*.

$$\sum_{\substack{q \in r \\ p \neq q}} v_{pq} \left\{ Y_{kp} \left(\sum_{\substack{m \in K^{\bullet} \\ m = k}} Y^{T}_{mq} w_{km} \right) \right\} \quad r \in R^{\bullet} \quad k \in K^{\bullet} \tag{6}$$

The third term computes the fraction of total communication due to the communication of process k assigned to processor p with processes m assigned to other clusters:

$$\sum_{\substack{p \in r \\ q \in s \\ r \neq s}} v_{pq} \left\{ Y_{kp} \left(\sum_{\substack{m \in K^{\bullet} \\ m \neq k}} Y^{T}_{mq} w_{km} \right) \right\} \quad k \in K^{\bullet} \quad r,s \in R^{\bullet} \tag{7}$$

The scaling factors and weights that are part of these equations are not shown. As with passive object association, the values ascribed to proximity relations are such that the maximal values of these equations are ascribed to the best assignments.

Two properties of the equations 5, 6, and 7 should be noted. First, these metric functions are quadratic in Y, which is a zero-one decision variable. As a result, the process assignment problem is also NP-complete. Second, three terms exist in the composite function because the Cm* architecture exhibits a three-level hierarchy in memory access. A straightforward extension of this metric function for a network architecture would use "n" terms, each term representing one level in the hierarchy of memory access.

5.1.3. Discussion of the Metric Functions

In this section, we discuss the changes that were made in the metric functions when they were used within TASK. In addition, the limitations of these metric functions are reviewed.

The metric functions presented here will be used by the heuristics discussed in the next section. However, during our experimentation with those heuristics, the metric functions were changed in several ways. The first change consisted of the addition of weights and scaling functions. For example, the values of the relations w_{km} in equation 4 were increased while the values of d_{ik} in the same

equation were decreased. As a result, the values of d_{ik} could not dominate the values of the metric function. The second change consisted of the introduction of rankings of computer modules with respect to their relative location or size. These rankings were employed to prefer certain computer modules with respect to placement and assignment. For example, we used weights within equations 5, 6, and 7 that led to preferences with respect to process assignment. Specifically, processes were assigned to different computer modules within a single cluster before they were assigned to computer modules in different clusters. As a result, processes executing in parallel will communicate by means of efficient intracluster memory accesses as long as processors are available in a single cluster. We note that ranking techniques are quite common in heuristics used in related mapping problems [112]. The third change consisted of the inclusions of the variables N_i and Q_k into the metric functions so that eligibility sets are taken into account by the tailoring heuristics. Last, we experimented with a metric function that expressed the total completion time of a task force in terms of individual process completion times. In the case of this metric function, the objective was to minimize completion time regardless of interprocess communication. The resulting tailoring decisions tended to maximize the degree of parallelism in all cases observed.

The minimization of the metric functions should result in tailoring decisions that act in accordance with the directives specified in the TASK program. However, recall that proximity directives can also express constraints concerning assignment and placement. It is not always possible to formulate metric functions whose optimization results in the fulfillment of tailoring constraints. Furthermore, if the satisfaction of constraints is synonymous with optimal metric function values, then constraints are not necessarily fulfilled by heuristics which are not guaranteed to find optimal metric function values. If constraints must be satisfied, a separate phase must be added to the heuristics presented in the next section. In this phase, called the *constraint satisfaction phase*, the constraints that are not fulfilled by the tailoring heuristics are acted upon. In addition to the implementation of this phase, we attempt to reduce the number of constraints violated by tailoring heuristics by introducing biases into metric functions. A sample bias of a metric function is one in which the *Diff-Cm* values of the relations

c_{ij} or w_{km} are increased so that these relation values are chosen when attempting to gain high metric function values.

While formal validations of the mathematical formulation in this section are not feasible within the limits of current experimental results, three specific benefits of the formulation can be identified. First, the model instances and the metric functions in this section are easily changed to describe other software and other distributed architectures (see Chapter 3 for a discussion of this topic). However, as previously stated, our formulations are static in nature because they rely on "snapshot" information concerning the task force and the distributed architecture. Second, since the task force mapping problem is formulated as a standard integer programming problem, it is straightforward to understand the mapping problem's complexity. In addition, the heuristics developed for TASK are easily compared with heuristics developed for related problems. Third, since the development of solution heuristics is separated from the formulation of metric functions, both can be changed independently of each other. As a result, experimentation with different combinations of metric functions and heuristics is straightforward.

5.2. Tailoring Heuristics

Several properties of the task force mapping problem prompt us to choose heuristic rather than optimal tailoring procedures. Our reasons for this choice are the same as those of Gylys [61] who extensively studied optimal algorithms and heuristic procedures for a related problem. Specifically, since the formulated zero-one integer programming problems are NP-complete, optimal solutions are hard to obtain. Furthermore, by necessity, the software and hardware instances used by TASK contain only partial, sometimes inaccurate, information concerning the task force and its execution environment. As a result, it is unreasonable to seek optimal solutions. Note that even the collection of information by a system monitor only offers improvements and not guarantees of accuracy. Another reason to choose heuristics is that optimal solutions of the mathematical formulation need not correspond to optimal solutions of the stated tailoring objective because the optimization of the

presented metric functions does not necessarily result in optimal speedup tailoring decisions. Last, fast heuristics are preferable to slow optimal procedures because such heuristics can even be used during the execution of a task force, where the cost of tailoring must not outweigh the gains derived from tailoring.

The heuristics presented in the remainder of this section are similar to heuristics used elsewhere [112, 61]. Since such heuristics typically attain solutions that are fairly close to optimal, we do not investigate the quality of the solutions attained. Instead, we demonstrate the practicality of using heuristics in a system like TASK, namely it is shown that the heuristics are sufficiently fast and that the space and time required for their execution are not unreasonable.

5.2.1. The Heuristics Used In TASK

To develop the heuristics that follow, three commonly used design techniques are employed. First, the heuristics are *greedy heuristics*, namely they do not backtrack to test alternatives once a decision has been made. Second, the software and hardware objects are ranked by importance and ordered by size, ensuring that certain objects are associated before others. For example, large processes are assigned before small processes, and large processes are assigned to computer modules containing large amounts of memory. As a result, large computer modules are effectively used. In addition, if a metric function ascribes equal values to several associations of a specific passive object with processes, then the passive object is associated with the smallest process. The purpose is to balance the process sizes. The third technique, clustering, has already been discussed.

The heuristic procedures for passive object association and active object assignment are outlined as programs written in an abstract high-level language. The language constructs in these programs are italicized. Within these programs, the metric functions are referred to by their equation numbers.

Passive Object Association:

Sort all passive objects by size, largest first

For each passive object $i \in I^*$:

Compute $\max_{m \in K^*}$ {Equation 4}
resulting in a set of process indices, where each index represents one of the "best" processes with which i should be associateded

If the resulting index set contains one process, k,
then set $X_{ik} = 1$

Else first compute the total size of the passive objects that have already been associated with each of the processes in the index set; select the smallest of those processes, m, and then set $X_{im} = 1$

Continue with the next passive object

Process Assignment:

Sort the processes by their size, largest first

For each process $k \in K^*$:

Compute $\max_{p \in P^*}$ {Equation 5 + Equation 6 + Equation 7}
resulting in a set of processor indices, where each index represents one of the "best" processors to which k should be assigned

Given the current assignment of processes to processors, compute which processor in the index set has the largest amounts of unused space

If there is such a processor, *then* pick that processor, say p

Else since no space is left in any of the processors in the index set, choose the processor in the index set with the least size memory constraint violation, say p

Set $Y_{kp} = 1$ and *continue* with the next process

It is clear that these heuristic procedures associate passive objects with "best" processes and assign processes to "best" processors, where "best" associations and assignments are determined

by the maximum values of the metric functions. Some extensions are required when these procedures are used within TASK. One extension concerns the *eligibility sets* of processes (see Chapter 4). Eligibility set restrictions can be formulated as restrictions on the set of processors tested by the association or assignment heuristic. Specifically, rather than computing a maximum over all processors $p \in P^*$, we compute:

$$\max_{p \in Q_k} \{\text{Equation 5} + \text{Equation 6} + \text{Equation 7}\}$$

Another extension concerns memory constraints, which may be exceeded when large processes are assigned to small computer modules. This extension is implemented in two ways. First, since the size of each process is known, this size can be matched against the sizes of the computer modules contained in the index sets used in the procedures above. However, if processes are too large to fit into any computer module in those index sets, then different actions must be taken. In such cases, a *passive object shift* procedure is run after passive object association and active object assignment have been performed. In this procedure, passive objects are first shifted among processes in an attempt to achieve better process fits, and are then directly placed into memory if the processes cannot be fitted. *Component of* relations and *type* information are used to select the specific objects that should be shifted. For example, since the effects of shifting objects of type `Mailbox` are typically not significant with respect to task force performance, such objects are shifted first.

Two additional tasks are performed by the passive object shift heuristic. First, the procedure tests whether any associations of passive objects with processes violate the eligibility sets of the passive objects. Such violations are removed by shifting the passive objects, first to other processes and then directly to computer modules. The second task concerns the manner in which shifting is performed. Namely, passive object shifting provides a chance to review the decisions made by the greedy heuristics. Decisions are reviewed by recomputing the values of metric functions each time an object is shifted.

Although object shifting is an integral part of tailoring in TASK, its detailed description is elided to

curtail the length of this discussion. Instead, we summarize the actions of the passive object shift procedure:

- memory constraint violations are detected and eliminated by shifting passive objects between processes or into computer modules to which no processes are assigned;
- the placement constraints imposed by the passive objects' eligibility sets are fulfilled;
- the decisions of the greedy assignment heuristics are reviewed by recomputing and acting upon metric function values during shifting.

5.2.2. TASK, a Testbed for Tailoring Heuristics

The TASK system is well-suited for further experimentation with tailoring. For example, newly acquired knowledge regarding good tailoring practice can be integrated into TASK in the forms of updated metric functions or tailoring procedures. Such integration can be performed without extensive changes because tailoring procedures are written to maximize the values of metric functions, regardless of the specific metric functions used. Since it is straightforward to use different metric functions, it is equally straightforward to pursue different tailoring objectives. In addition, the particular tailoring objective that is being pursued is irrelevant to the design and implementation of tailoring heuristics; experimentation with metric functions and tailoring procedures can be performed independently of each other.

The TASK system also facilitates experimentation with different ways of determining proximity relation values. Specifically, since relations are stored in an explicit form within the TASK compiler, they can be assigned values more than once. As a result, relation values can be statically derived from proximity directives, and relation values can be dynamically by a task force monitor. In this manner, tailoring decisions that are made based on proximity directives can be revised after task force execution. We note that the TASK system has not yet been interfaced to a task force monitor. However, the system already provides useful information to task force programmers. On demand, TASK outputs the placement and assignment decisions that were made, the proximity information based upon which those decisions were made, and statistics concerning the current task force.

5.2.3. Experiments with Tailoring Heuristics

Programmers expect compilers of higher level languages to generate code comparable in quality to code written by hand. Similarly, programmers of Cm* expect tailoring heuristics to make reasonable assignment and placement decisions based on the specified directives. We cannot determine how close to optimal the automatic tailoring decisions are. However, in the following we investigate the behavior of the heuristics with respect to their faithfulness regarding the proximity directives stated in TASK programs. The metric used to measure such faithfulness ascribes values to the match of tailoring decisions with the relations d_{ik}, w_{km}, and c_{ij}.

To investigate the faithfulness of tailoring heuristics, several typical situations encountered by tailoring heuristics are discussed[22]:

- task forces mapped to a system with ample resources;
- task forces mapped to a system with insufficient resources;
- task forces that are constrained by explicit selections;
- task forces that are constructed without proximity information; this case arises if naive users do not specify directives, whereupon the TASK system uses built-in tailoring knowledge.

Tailoring with Ample Resources

In programming practice on Cm*, the construction of experimental task forces that require only a few resources of Cm* is commonly performed. Consider the construction of a PDE task force with one coordinator process, three servers, and a grid with 11 partitions of size 2K words each. In this example, assume that a single cluster of Cm* containing 4 processors and large amounts of memory is available.

In addition, the following directives are specified in the TASK program:

```
SameCm (Stack, Code)  -- appearing in both function templates
DiffCm ((i=0..10) PDEGrid [i])
DiffCm ((i=0..2) Server [i], Coordinator)
```

[22] Several different task forces, including the PDE task force, were employed in this analysis.

With these directives, a programmer is directing server processes to execute in parallel and to spread the grid partitions across multiple computer modules.

When TASK's tailoring heuristics are run, the passive object shift procedure is not executed, since sufficient amounts of memory are available and since the eligibility sets of task force objects are not restricted. The following decisions are made:

- the grid partitions, all of which have equal proximity values with respect to each of the processes (equal values of d_{ik}), are associated with the first process listed, which is the coordinator process;
- the code and stack objects, all of which have high proximities to particular processes, are associated with the appropriate processes;
- since there is a sufficient number of computer modules, all processes are assigned to different processors;
- since large amounts of available memory exist in the selected cluster of Cm*, even the large coordinator process fits into a single computer module.

A comparison of these decisions with the stated directives shows that the proximity relations between processes and passive objects (d_{ik}) and the proximity relations between processes (w_{km}) are followed faithfully, whereas the proximity relations between the grid partitions (c_{ij}) are not followed. This lack of faithfulness with respect to c_{ij} is explained by the form of the metric function Equation 4. In this equation, the proximity relation values c_{ij} are summed over all passive objects j associated with each process m. Consequently, once any process m has been given a passive object, given that the values of d_{ik} and w_{km} are equal, this process will "draw" further passive objects by dominating within the metric function.

The resulting lack of faithfulness concerning the relation c_{ij} is tolerable in this example, because the total number of accesses to the grid partitions is not exceedingly high. Therefore, the total number of accesses to the single memory unit containing the partitions is not excessive. However, this lack of faithfulness would not be tolerable if grid partitions were accessed more frequently, because memory contention would significantly increase the time required for each access [29]. To increase

the faithfulness with respect to c_{ij}, functionality is added to the passive object shift procedure. An optional bias is introduced into the metric function Equation 4. Specifically, instead of computing Equation 1, we compute the conditional sum:

$$\sum_{j \in I^*} \text{violations}\,(c_{ij}\, X_{jm}) \quad i \in I^*, m \in K^* \tag{8}$$

In this equation, particular values of c_{ij} are included into the sum only if the inclusion of these values does not violate the constraints expressed by the relation. For example, given a `DiffCm` directive between objects i and j, the value of c_{ij} is included into the sum only if i and j are being placed into different computer modules.

The exercise of this option in the passive object shift procedure improves faithfulness. To demonstrate this improvement, we display both faithless and faithful decisions in figure 5-1. In one set of decisions, the conditional metric function is used, whereupon the `DiffCm` directive is followed faithfully. The other set of decisions demonstrates faithless behavior with respect to the relations c_{ij} between grid partitions. However, violations with respect to c_{ij} still occur in this example, because the grid partitions cannot be spread across more computer modules than are available. Note that both sets of decisions are faithful with respect to the values of d_{ik}: code and stack objects are always placed with the appropriate processes.

At this point, one characteristic of the tailoring procedures can be noted. Specifically, the association heuristic for passive objects results in faithlessness with respect to the c_{ij} relation values `DiffCm`, unless the equation 8 is used in the metric function.

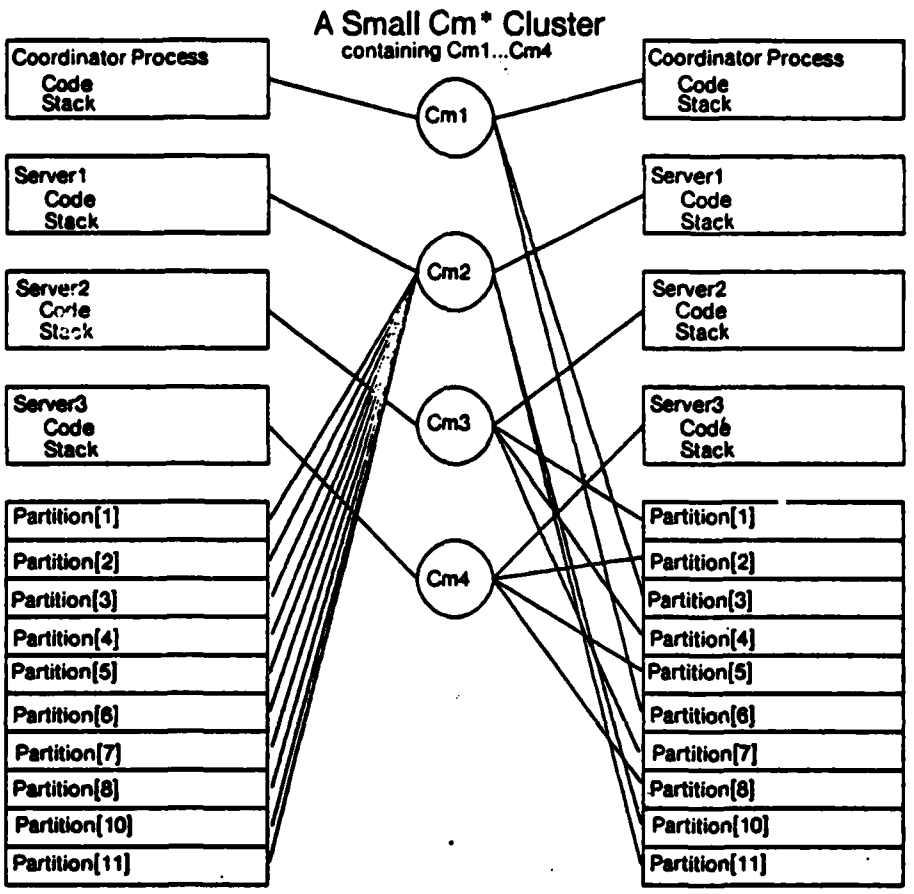


Figure 5-1: Increasing Faithfulness by means of a Passive Object Shift Procedure

Tailoring with Insufficient Resources

Situations in which there are insufficient or barely enough resources to complete the task force construction are commonly encountered in small configurations of Cm*. To use the available resources to the best degree possible, the passive object shift procedure reduces the violations of memory constraints by shifting passive objects among processes or by placing passive objects into processors to which no processes have been assigned.

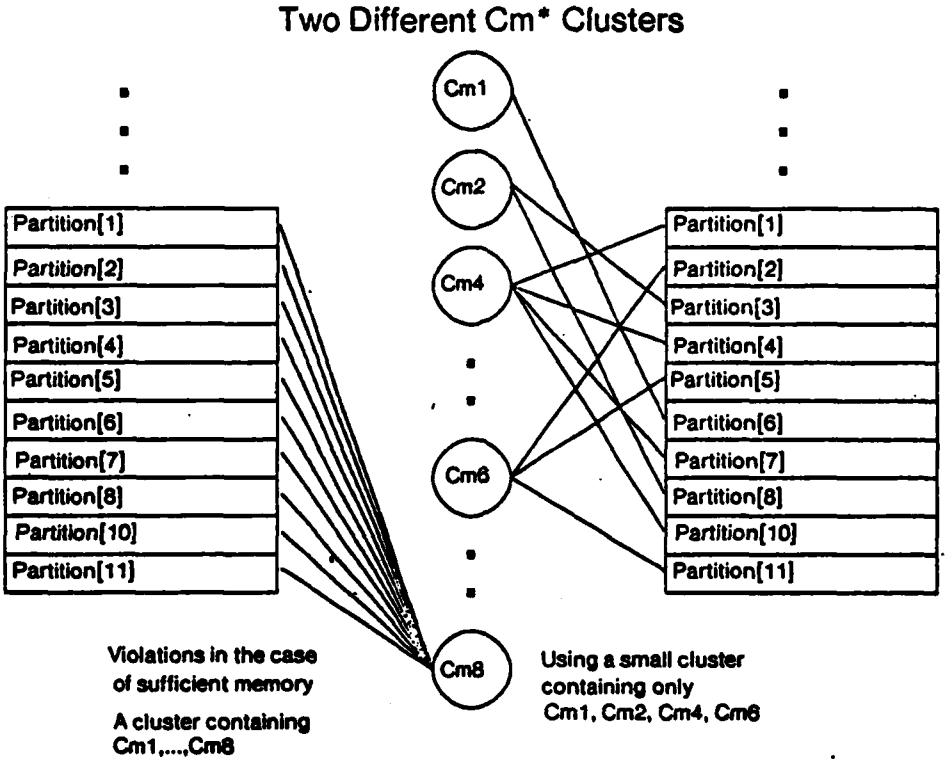


Figure 5-2: Increases in Faithfulness Due to a Lack of Resources

Comparative experiments with a standard size versus a very small Cm* cluster demonstrate the usefulness of the passive object shift procedure. In figure 5-2, two sets of decisions made by the tailoring heuristics are displayed. In one set of decisions, the larger cluster is used so that the passive object shift procedure is not executed. In this case, the entire grid is placed into one memory module. Clearly, this demonstrates lack of faithfulness with respect to the relations c_{ij}. In the other set of decisions, a lack of available memory improves faithfulness with respect to c_{ij}. Note that the passive object shift phase attempts to balance the sizes of processes by exhibiting a bias in object association for the processes that are smallest. This "switching" effect can be observed with respect to Cm2, Cm4, and Cm6 (see figure 5-2). Cm1 is not involved in this switching effect because this Cm has

already been filled to capacity with the (elided) coordinator process.

Another characteristic of the tailoring heuristics can be noted. Specifically, scarce resources can improve faithfulness with respect to c_{ij} so that the suggested alteration to the metric function's equation 1 (see equation 8) may not be necessary. However, the suggested alteration is clearly shown to reduce the number of memory violations.

Task Force Tailoring and Selections

Eligibility sets are easily accommodated within the framework of the presented metric functions and heuristics. Regarding the assignment of processes to processors, the equations 5, 6, and 7 are simply altered to select the best process from the processors in each process' eligibility set. However, the eligibility sets of passive objects must be accounted for in the passive object shift procedure.

The existence of eligibility sets typically increases the speed with which tailoring decisions are made. This is due to a resulting reduction in the number of choices that must be tested. The speed of tailoring is increased further if the processes and passive objects whose eligibility sets contain exactly one resource are preprocessed and are not considered by the tailoring heuristics. However, while eligibility sets are beneficial with respect to the speed of tailoring, their existence can severely degrade the heuristics' faithfulness with respect to the relations w_{km} and c_{ij}. Consider the case in which each cluster of Cm* contains one computer module with an attached disk. If each of two processes is explicitly assigned to a processor with an attached disk, and if both processes are related as `DiffCm`, then the explicit assignment stated causes faithless behavior with respect to the relation value `DiffCm`.

Task Force Tailoring without Proximity Directives

Naive users of Cm* typically wish to construct task forces while acquiring the least amount of information possible about Cm* and TASK. At the same time, they expect their task forces to exhibit reasonable runtime performance.

TASK's tailoring heuristics accommodate such users by using built-in knowledge concerning the performance-related properties of Cm*. In addition, the TASK compiler uses the *type* and the

component of information in the execution blueprint of the TASK program to derive appropriate default proximity relations. Instead of listing the relations that are derived by the TASK compiler and then used by the tailoring heuristics, we elaborate upon the defaults used for the sample PDE task force. If no directives are specified, proximity relations equivalent to the following directives are derived by the TASK compiler:

```
SameCm (Code, Stack) -- appearing in each function template
DiffCm ((i=0..2) Server [i], Coordinator)
SameCluster ((i=0..10) Grid [i])
```

Note that tailoring heuristics that faithfully carry out these directives will make decisions with which linear speedup is attained in the executing task force (see Chapter 3).

In the example above, appropriate tailoring decisions are made even if programmers do not state proximity directives. This is not true in general. For example, the directives above do not result in good performance for a two cluster version of the same PDE task force, since the `DiffCm` directive suggests that all processes remain in the same cluster. In cases like these, one can improve TASK's tailoring heuristics by introducing biases into the metric functions rather than using the default directives. For example, the equations 5, 6, and 7 could be biased to assign processes to computer modules in the following order: first, to different computer modules within a single cluster, next to computer modules in different clusters, and last to the same computer module. However, biases like these cannot replace proximity directives, since each set of biases typically represents one particular directive.

The Cost of Tailoring Heuristics

The heuristics implemented as part of the TASK system are not optimized for their use of time and space. In the current implementation of the TASK system, space-inefficient matrices are used to encode the relations d_{ik}, w_{km}, and c_{ij}. In addition, time-inefficient linear list structures that contain pointers to task force objects are manipulated by the heuristic procedures. Nonetheless, all tailoring problems described in this section were solved in less than 1/2 second of CPU-time on a KL-10. Larger problems containing 20 or 30 processes and up to 60 data components consumed ap-

proximately 1/2 to 2 minutes of CPU-time.

Conclusions of the Experimentation

We have demonstrated the feasibility of integrating tailoring heuristics into the TASK system. Furthermore, the actions of tailoring heuristics in typical situations encountered during tailoring have been shown to be reasonable. Reasonable tailoring actions are attained because heuristics are faithful to proximity directives, while also maximizing the metric functions stated. Tailoring heuristics without the passive object shift procedure are faithful with respect to proximity relations between processes and between processes and passive objects. The passive object shift procedure that uses equation 8 achieves faithfulness with respect to proximity relations between passive objects.

The selections that are stated in TASK programs are accommodated both during the assignment of processes to processors and in the passive object shift procedure. If neither selections nor proximity directives are stated in TASK programs, some measure of reasonable performance is guaranteed by the inclusion of biases into metric functions and by the automatic derivation of proximity relations. The use of tailoring heuristics is generally not too time-consuming. This is partially due to the fact that passive objects are clustered around processes rather than placed individually.

5.3. Extensions of this Research

At this point in time, considerable information exists concerning the appropriate use of multiple processor systems. Therefore, it is no longer sufficient to use such information in an adhoc fashion. Specifically, one cannot rely on the availability of expert programmers who can make judicious assignment or placement decisions. Expert programmers are not needed in TASK because tailoring knowledge is easily integrated into the TASK system. Given the functionality of TASK and of Cm*, unique opportunities exist for future research:

- tailoring can be performed with different objectives, which are expressed by different metric functions;
- alternative tailoring heuristics or metric functions can be developed to reduce the time and space required for tailoring;

- good tailoring practice can be integrated into the system in a stepwise fashion: by biasing cost functions, by adding functionality to passive object and process assignment, and by adding functionality to the passive object shift procedure;

- information that is fed back from a task force monitor to tailoring heuristics can be used to improve the quality of tailoring decisions;

- metric functions and heuristics can be developed for architectures other than Cm*.

6. Conclusions and Future Research

Writing multiple processor applications of substantial size requires considerable programming expertise [79]. Within the TASK system, we explore two approaches to reducing this expertise. First, software is constructed such that programmers need not know any detail concerning the programming tools involved in construction, such as linkers and loaders. Second, programmers are assisted in tailoring to the multiple processor machine used for execution of their application programs. Tailoring is performed such that programmers need not know unnecessary hardware detail.

In the remainder of this chapter, multiple processor software development and tailoring are discussed in turn. The main results of our research are reviewed and future research is suggested.

6.1. Software Development

Several conclusions concerning the development of multiple processor software can be drawn from the design and implementation of the TASK system and language. One conclusion is that the abstractions typically offered by uniprocessor programming environments can also be used to construct executable software for multiple processor systems. Specifically, as demonstrated within TASK, programmers can design and implement multiple processor programs as collections of separable program modules, where each module is a unit of abstraction [164] or a unit of functional decomposition [130]. From these modular decompositions, which are familiar to most application programmers, the less familiar software descriptions required for multiple processor systems--sets of concurrent, communicating processes called task forces--can be derived by use of simple, manual and automated procedures.

Although the modular decomposition of an application program is seemingly different from the derived executable task force, a single model of software can describe both. Namely, each description of an application program can be represented as a set of related objects, where relations differ

depending upon their intended use. We call such program descriptions the blueprints of a program. Two different blueprints of TASK programs are used in the TASK system. The logical blueprint encodes the structure of the modular decomposition of an application program. Therefore, a logical blueprint of a TASK program is analogous to the abstract specification of an object in an abstraction language [164]; this blueprint typically remains unchanged across experiments with the executable task forces constructed from it. However, analogous to the different implementations of an abstract object, the blueprints that can differ from one experiment to another are the detailed descriptions of the executable task force; these blueprints are called the execution blueprints of a TASK program.

The TASK system assists programmers in deriving execution blueprints from logical blueprints. Specifically, substantial parts of execution blueprint contents are specified by defaults generated by TASK, thereby suppressing information not important to programmers. Furthermore, specific, useful variations in deriving execution blueprints from logical blueprints are supported. Useful variations in the construction of an executable task force are those that facilitate typical experiments performed with multiple processor software. For example, to perform experiments measuring the dependence of task force performance on the number of task force processes executing in parallel [84], task force construction can be varied by varying the replication of processes in the executable task force.

In TASK, variations in task force construction are easily implemented only if the algorithmic code executed by a task force is not affected by those variations. However, future research in multiple processor programming environments should also consider construction variations that require alterations to algorithmic code. For example, if the communication protocols used by the algorithmic code were known to the programming environment and could be automatically changed, then the specific means of communication employed by task force processes could be determined automatically. In this fashion, an executable task force could be customized to use different communication mechanisms in different execution environments. For example, in a multiprocessor shared memory could be used for interproc . communication, whereas ports or mailboxes would be used in a computer network.

The customization of software to different execution environments would be facilitated if task force blueprints were displayed graphically, and if the task force components available for inclusion into a blueprint were displayed in menus describing relevant component characteristics. In addition, intelligent display aids could assist programmers in comprehending and analyzing the complex structures of task forces that consist of a large number of different components and component relations.

Menus of components to be included into blueprints require that self-contained units of description of task force components are available. In TASK, such units are called templates, and task forces are developed as compositions of programmer-defined templates. Therefore, an extension of TASK supporting component menus could simply use predefined or standardized templates. We note that a "skeleton" process [149] plays a similar role in the development of software for the STAROS system.

Templates can be employed for both static and dynamic component construction. Statically, templates defined in TASK programs can be repeatedly instantiated, thereby causing the construction of multiple task force components. Dynamically, the instantiation of a template of type process causes the creation of a process in the executable task force. As with process templates, TASK can be extended to accommodate templates of any type during task force execution. Therefore, the incremental, static or dynamic construction of a task force can be implemented with ease.

In the TASK system, programs are written in two languages: the TASK language and an algorithmic programming language. Several benefits result from this separation of languages. First, since task force structure is described with the TASK language, the algorithmic language is only used to write the code executed by task force processes. Therefore, the algorithmic language need not be overloaded with multiprocessing features [72]. Similarly, the TASK language exhibits little complexity and is therefore, easily mastered. Second, since logical structure and algorithmic code are specified separately from each other, programmers are forced to consider each in turn. It has been argued that such separation of programmer concerns is beneficial. Third, the extension of one language by another can preserve compatibility, whereas the addition of multiprocessing features to a language

cannot. However, if two languages rather that one augmented language are used, language interfaces must be constructed to avoid inconsistencies between the versions of the application program expressed in each language. Such an interface does not exist in TASK, where only the names of task force components are shared between the two different language descriptions of a task force. As a result, TASK program attributes cannot be controlled by Bliss code, and Bliss code cannot be affected by attributes of TASK programs. However, certain extensions of the language interface are straightforward. For example, TASK could be easily extended to check the types of parameters passed between the different modules and module functions declared in TASK programs.

Details of the TASK language and its specific syntax are peripheral to this thesis. Instead, the important attributes of the TASK language are the distinction of structural task force descriptions from the algorithmic code (see the discussion in the previous paragraph) and the usefulness of the language for a variety of hardware and software. Specifically, reformulations of the TASK language can describe distributed hardware [82] or can describe software written to execute on computer networks. Furthermore, program blueprints and therefore, reformulations of the TASK language can be used to represent multiple processor applications at any level of detail required by programmers. For example, in Chapter 3 of this thesis, an entire operating system is described by a blueprint characterizing the system's functional decomposition. Each unit of description of this blueprint consists of multiple components of the task force blueprints discussed above. We note that program blueprints can also be used to describe multiple processor applications at execution time (see Chapter 3 and [144]).

6.2. Task Force Tailoring

The ease of variation of task force construction exhibited by the TASK system facilitates the experimentation with executable task forces. However, such experimentation remains difficult unless programmers are assisted in tailoring an executable task force to its execution environment, where tailoring is defined as the allocation of hardware resources to task force components. In this thesis,

assistance in task force tailoring consists of the partial automation of tailoring. The automatic tailoring procedures designed and implemented are not the optimal algorithms published in the theoretical literature because such algorithms are too slow. Instead, fast heuristics act in accordance with our current, best knowledge concerning tailoring for the Cm* hardware. Since this knowledge is subject to change, tailoring procedures permit the inclusion of new knowledge. Specifically, tailoring policies can be altered or exchanged and multiple objectives are accommodated by tailoring procedures, so that programmers can perform tailoring with different policies and objectives (see Chapter 3).

Since our current knowledge concerning good tailoring practice is insufficient and since tailoring decisions often depend on the specific hardware used for task force execution, tailoring has not been automated completely. Instead, programmers provide tailoring assistance by stating resource directives that express the proximity of task force components as frequency of access relations between those components. Resource directives need not be stated for individual task force components. Instead, directives relate sets of task force components.

Two interpretations of resource directives have proven useful: their interpretation as expressions of preferences concerning resource allocation or their interpretation as expressions of constraints. Preferences are suggestions that need not be honored by tailoring procedures, whereas tailoring constraints must always be honored. Therefore, preferences can be used as hints based on which tailoring choices can be made freely, whereas constraints can be used to exert explicit control on resource allocation, if desired. Programmers can manually check and improve TASK's resource allocation decisions.

The formulation of tailoring in terms of resource directives, tailoring objectives, and tailoring procedures is based on an abstract model of multiple processor software and hardware, called the proximity model. We use this model to show that TASK is easily extended to tailor software for a variety of multiple processor architectures. Similarly, the components of the TASK system that are implemented according to this model can be used in extensions of our research:

- to tailor software for architectures other than Cm*;
- to investigate the effects of alternative tailoring objectives and tailoring procedures;
- to vary the degrees of programmer interaction in tailoring:
  - by specifying few or many resource directives;
  - by accepting or overruling TASK's automatic tailoring decisions;
  - by including alternative tailoring procedures into the system (the current TASK system permits programmers to choose among a variety of tailoring procedures).

The notion of tailoring could be generalized in several ways (see Chapter 3). One generalization involves the use of knowledge concerning task force behavior during execution. For example, the quality of tailoring decisions could be improved if the actual control flow in the executing task force were known. Specifically, if two processes cannot execute in parallel, they need not be assigned to two different processors. Another use of knowledge concerning the behavior of an executing task force is a comparison of the actual access rate of a process to an object with the access rate estimated by a user-defined resource directive. If such comparisons were performed, TASK's tailoring decisions could be based on information superior in accuracy to the estimates stated by programmers. The implementation of a task force monitor and of an interface between TASK and the monitor play a critical role in such an extension of the TASK system.

An extension of our investigation of tailoring is the consideration of the tradeoffs between static and dynamic tailoring. Since a task force is subject to change during execution, dynamic tailoring is required to supplement or improve the decisions made statically. The feasibility of dynamic tailoring depends on the speed with which tailoring decisions are made and on the speed with which the information required for dynamic tailoring is collected. Since the tailoring procedures used in TASK can be sped up further by limiting the amount of information that is processed, dynamic tailoring appears feasible.

Appendix A
The TASK Language

Multiprocessor computers offer the potential advantages over uniprocessors of enhanced reliability and cost-effective performance.[23] To this end we construct software in the form of *task forces*—collections of cooperating, communicating processes, which use system synchronization and communication mechanisms to solve a single problem. Such software is difficult to construct without special tools, due to the complex execution environment of the multiprocessor. In particular, those users who wish to capitalize on multiprocessors to attain their reliability or performance objectives cannot ignore system attributes that affect those objectives. Multiprocessors do not introduce qualitatively new difficulties in scheduling or process and data management. They do, however, complicate those aspects by increasing the number of options beyond that for a uniprocessor, hence increasing the number of decisions that must be considered.

A.1. The Goals of TASK

The BLISS language itself offers no facilities for managing task forces, nor do any existing multiprocessing languages incorporate features concerning resource allocation for task force components. The TASK language includes these facilities. TASK can be thought of as an extension to BLISS. Individual STAROS *modules* are written in BLISS, and the relationships among them, as well as their placement for resource-usage optimization, are specified using TASK. All programmed actions performed by modules after they have been loaded are written in BLISS. BLISS and TASK have been integrated to the extent that both are naming and manipulating the same objects.

TASK is a high-level specification language in which an author may specify the different initial

---

[23] The bulk of this appendix consists of text that has been adapted from a draft of a technical report, *The TASK Language Specification* [82]. Further information about the TASK language can be found in two technical papers [81, 83] and in this thesis.

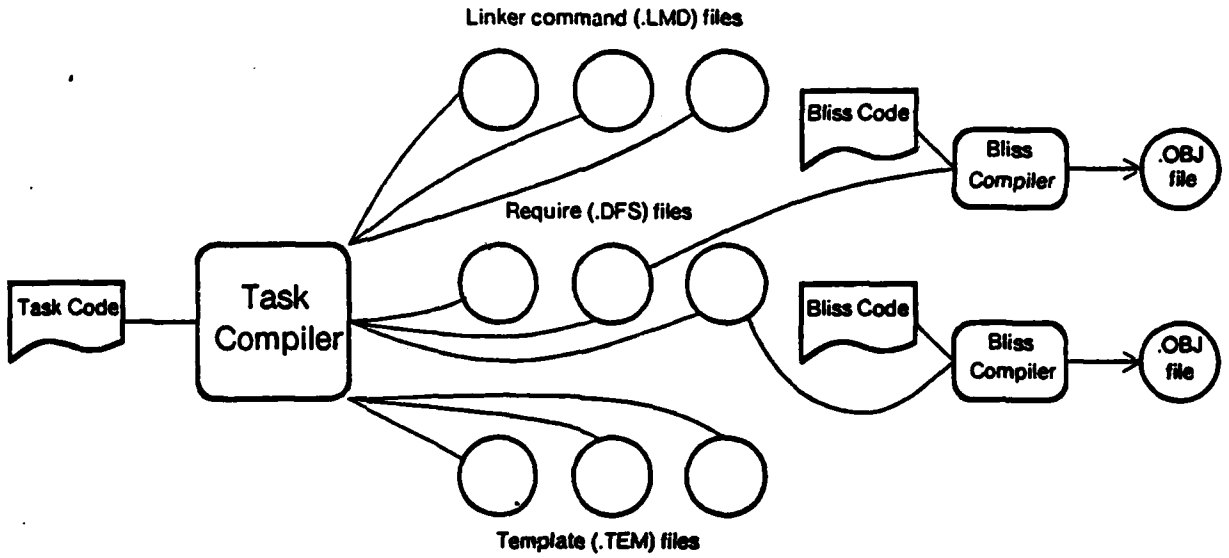


Figure 6-1: Compiling a Set of BLISS Programs and a TASK program

components that comprise his task force. TASK does not constrain how the task force behaves once execution commences; all facilities of STAROS are available to modify at run time a task force created according to a TASK specification. Using TASK, the programmer can specify objects to be created and how they are to be initialized, as well as certain relations between objects. As described in [149], STAROS programs are constructed in the form of *modules* which export one or more *functions* which may be invoked by code in other modules. Part of the purpose of TASK is to define the interconnection between modules by providing one module with capabilities for objects in another. We say that TASK furnishes *a programming environment* for constructing task forces.

TASK provides several ways to adapt a task force from run to run. Changes can be made in order to obtain better performance, or to accommodate variation in the physical configuration of Cm* or the input data. For instance, data might either be stored as a unit in a single computer module or *partitioned* among several objects in different Cm's. Data or processes might or might not be replicated.

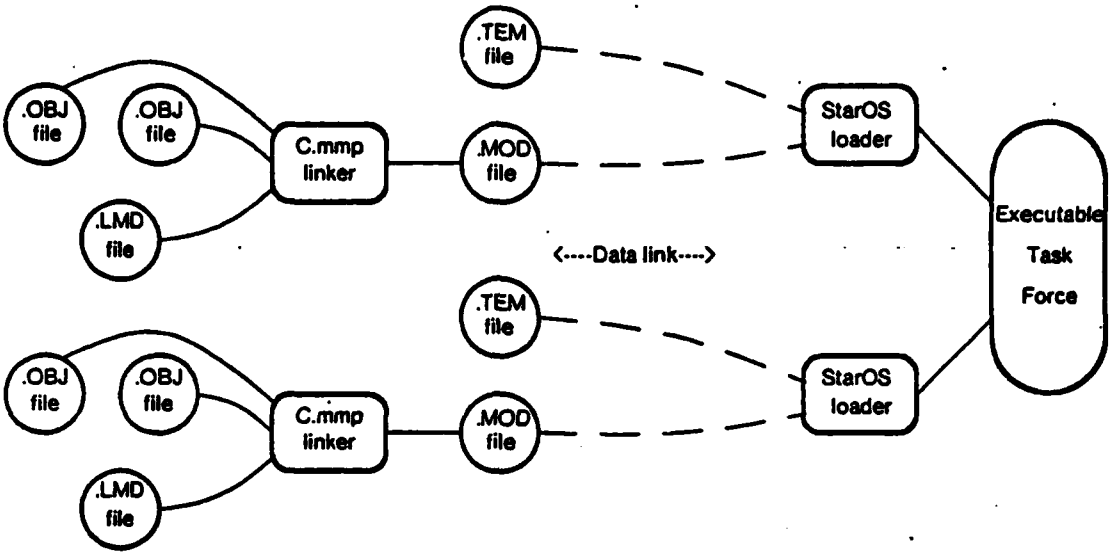


Figure 6-2: Linking and Loading a Task Force

TASK also provides facilities for controlling resource allocation. Task-force authors may express relationships between programs and data, or constraints on resource utilization. To do this, they specify

- *assignment*—which process should execute on a particular processor, and
- *placement*—which Cm should hold a particular unit of code or data.

A.2. The TASK Compiler

The TASK compiler runs on CMU-10A, not on Cm*. The services provided by the compiler are summarized below.

- T*ASK* describes a task force in terms of the abstractions provided by S*TAR*OS. It describes the modules that make up the task force, specifying for what objects the modules will have capabilities and what functions are exported by each module. It specifies an initial process structure for the task force by listing the processes that are to be created as soon as the task force is created, or "instantiated." (To *instantiate* means to create one instance, or example, of.)

- *TASK* generates directives for linking together the components of the task force. It writes a separate linker command file (.LMD file) for each module, as well as one which relates to the overall task force. Reading directives from an .LMD file, the C.mmp linker combines several relocatable object files produced by the BLISS compiler into a single file which may be shipped to Cm* and read by the STAROS loader.

  TASK generates definition files which contain the names of all modules, their functions, and other components. These files contain names declared in the TASK program that may be used in BLISS code. One BLISS `require` file is created for each module, one for each function exported by each module, and one for the task force as a whole, which is used for preparing a *task-force object*. The task-force object contains a description of the task force; it is used by STAROS but is not referenced by the user's programs.

  TASK also generates command files for the *STAR*OS loader. Again, one file is created for each module, and one additional file for the task force. These are the template, or .TEM files which are interpreted by the Loader in order to construct task force components and to allocate resources for the task force.

- *TASK* controls resource allocation for the task force, as specified by the user. It allows the user to relate components of a task force by expressing their performance characteristics. It also allows the user to specify the characteristics of Cm's where objects and processes should be placed.

The process of constructing a task force, from writing source code to running it on Cm*, is illustrated in Figures 6-1 and 6-2.[24]

A.3. Introduction to the Language

A TASK program consists of statements describing objects which contain data and code, relations among those objects, and their assignment to physical resources. The bulk of a TASK program is declarations; the one "executable" action is creating an object. Each such instantiation is realized by one or more invocations of the *Object Manager*. The types of objects that may be created are modules, processes, basic objects, stacks, deques, and mailboxes. A task-force author declares a template for each object that is to be created as part of the task force. A template specifies the type of an object and indicates its size, how it is to be initialized, and where it is to be placed. In many ways, templates are analogous to *type* declarations in languages such as Pascal and Ada.

---

[24] Figure 6-2 is slightly oversimplified; see the discussion on page 167.

Table 6-1: Expressions, Names, and Types in TASK

<Simple Name> ::= *<Unquoted String>*

<Simple Template Name>, *<Complex Template Name>* ::= *<Simple Name>*
Template names must be unique in their first five characters.

<Formal Parameter Name>, *<Var Name>* ::= *<Simple Name>*

<Comp Name> ::= *<Simple Name>*

<Function Name> ::= *<Complex Template Name>* . *<Complex Template Name>*
<Object Name> ::= *<Simple Name>* | *<Access Expr>*

<Path Name> ::= *<Simple Name>* . *<Path Name>* | *<Simple Name>*
<Object Path Name> ::= *<Object Name>* . *<Object Path Name>* | *<Object Name>*

<Keyword Name> ::=
<Obj Common Att> | *<Obj Add Att>* | *<Obj Special Att>* | *<Hard Att>*

<Var Type> ::= **String** | **Integer** | **Boolean**

<Expr> ::= *<Arith Expr>* | *<Quoted String>* | **True** | **False**

The rest of this chapter describes the syntax and semantics of the TASK language and includes a series of illustrative examples. The complete TASK grammar appears in Appendix B, but portions are repeated at appropriate places throughout this chapter. Most examples in the chapter are excerpts from a TASK specification for a task force that processes image data by performing filtering operations. An image is partitioned into slices, each of which may be processed almost independently of the others. Server processes that filter a slice need only cooperate with those server processes that filter an adjacent slice, A manager process is responsible for coordinating server actions by sending processing requests and additional data to servers via shared mailboxes. The manager process also handles the I/O associated with the image-processing task. The complete text of this example (albeit in slightly obsolete TASK), may be found in an early paper on TASK [81].

A.4. Templates and Instances

A template is analogous to a programming language *type* declaration while an *instance* is analogous to a variable of that type. A TASK program contains template definitions and instructions for instantiating object from those templates. Templates are comprised of a set of *attributes* which may be specified explicitly, or, if omitted, may be supplied by TASK from a set of default attribute values for the template. For example, the size of the object may be omitted from some template, while for another template resource-usage attributes may be omitted.

A template specifies several types of information:

- the type of an object,
- attributes of an object, such as its size, and where capabilities for it should be placed,
- instructions for building complex objects that have other objects as components, and
- resource-binding preferences.

A *simple template* gives directions for instantiating an object which does not have other objects as components. *Complex templates*, on the other hand, tell how to create modules and task forces, which have other objects (such as processes and modules) as components.

The syntax for templates is given in Table 6-2. In the syntax, three superscript symbols are used to denote different types of repetition:

- * means "zero or more repetitions of",
- + means "one or more repetitions of", and
- # means either zero or one instance of".

When items on a list are to be separated by some particular punctuation mark, the punctuation mark is indicated directly before the repetition character.

We will consider simple templates first. Examples of simple templates are—

```
Basic InputImage (Size=4K, Source=("Image1.obj", "Image2.obj"))
Mailbox (MsgType = "Capability", Size = 40)
```

Table 6-2: Syntax for TASK Templates

⟨Templates⟩ ::= *⟨Template⟩** EOF

⟨Template⟩ ::= {*⟨Complex Template⟩*|*⟨Simple Template⟩*}

⟨Simple Template⟩ ::=
⟨Simple Object Type⟩ ⟨Simple Template Name⟩(*⟨Actual Attributes⟩*)#
Some actual attributes may occur only within functions or modules. Hence there are semantic restrictions concerning which attributes may appear in simple templates that are not bound to a particular complex template.

⟨Complex Template⟩ ::= *⟨Task-Force Description⟩*
| *⟨Module Description⟩*
| *⟨Complex-Basic Description⟩*

⟨Simple Object Type⟩ ::= **Basic | Stack | Mailbox | Deque | Device**

The first template can be referred to as `InputImage`; it describes a basic object with a 4K-byte data part and no capabilities. The object is initialized from the two files `Image1.obj` and `Image2.obj`. Only the first 4096 bytes of these files will be recorded in the object when it is instantiated. The second template is for a capability mailbox that can buffer up to 40 messages.[25] Appendix C lists the creation parameters for all TASK object types. In particular, the interpretation of the `Size` parameter depends on which type of object is being created.

The *⟨Simple Template Name⟩*, if present, is associated with the template for future reference. A *⟨Template Name⟩* is not needed if the template is merely used once and never referred to again. Later examples will illustrate this.

The **New** construct is used to instantiate an object, either by specifying the template directly or by naming a previously defined template:

---

[25] If the number of "entries" for registered receivers is different from the number of messages, this may be specified by the syntax `Size` = (*number of registered receivers*, *number of messages*).

Table 6-3: Syntax for Complex Templates

<Module Description> ::=
Module *<Complex Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
<Function Description> +
<Resource-Usage Directives>

All module attributes must be specified at declaration time. Modules can only be components of task-force templates. Only one instance of a module can be constructed from a particular template.

<Function Description> ::=
Function *<Complex Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
<Resource-Usage Directives>

A *<Function Description>* can only be a component of a *<Module Description>*.

<Task-Force Description> ::= **TaskForce** *<Complex Template Name>* **is**
<Construction Description>
<Resource-Usage Directives>

A task force template can have no parameters. A *<Task-Force Description>* cannot be a component of another template. No more than one task force template may appear in one TASK program.

<Complex-Basic Description> ::=
ComplexBasic*<Complex Template Name>*(*<Formal Parameters>*)# **is**
<Construction Description>
<Resource-Usage Directives>

Components may have any type except those specifically excluded above.

```
New Mailbox (MsgType = "Capability", Size = 40)

New InputImage
```

The first example of the **New** construct will result in creation of a mailbox able to buffer 40 capability messages. The second example of **New** refers back to the simple template defined earlier. The body of the template named `InputImage` is textually substituted for the name `InputImage` (defined in the previous example). A basic object of 4096 bytes will be created, and initialized to hold up to 4096 bytes of the source files named `Image1.obj` and `Image2.obj`.

Table 6-4: Syntax for Construction Descriptions

<Construction Description> ::= **Construct** (*<Component>* ;*)

<Component> ::= *<Comp Name>* : *<Operation>*
| *<Iteration>* : *<Operation>*
| *<Expanded Iteration>* : *<Operation>* [not yet implemented]

<Operation> ::=
New {{*<Object Type>* (*<Actual Parameters>*)#}
| *<Template Name>* (*<Actual Parameters>*)#)}
| **Reserve** {{*<Object Type>* (*<Actual Parameters>*)#}
| *<Simple Template Name>* (*<Actual Parameters>*)#}
| **Ref** *<Object Name>* (*<Actual Attributes>*)#
Only special attributes should be used here.
| **Use** *<Object Name>* (*<Actual Attributes>*)#
| **Process** *<Function Name>* (*<Actual Parameters>*)
Note: returns a mailbox. Semantics: Processes may be created only within the body of the module defining their function
| **Name** (*<Actual Attributes>*)#
Only the **Window** attribute should appear here.

A.5. Complex Templates

Only task-force, module, function, and (complex) basic objects are specified by complex templates (see Table 6-2). The *<Construction Description>* specifies names and templates for each component object. The *<Construction Description>* also defines names for component objects that will be created by the task force as it is executing. The syntax of a *<Construction Description>* is given in Table 6-4.

When an object is instantiated, its component objects are also instantiated, either by manufacturing new objects or by acquiring the appropriate capabilities. The **New** construct indicates that an object is to be created. Similarly, the **Process** construct indicates that a new process is to be created to execute a specific function. (The process does not begin execution until it is scheduled, however.) A

component object may also be a parameter—in particular it may be a shared object—in which case it is named in the parameter list. The **Ref**, **Use**, **Name**, and **Reserve** constructs are explained later.

As an example of a complex object, let us consider a *module template* that makes use of associated mailbox and basic templates:

```
Mailbox Communicate (MsgType = "Capability", Size = 10)
Basic InputImage (Size = 4K, Source = ("SeaScape.obj"))

Module Server is
Construct (
                MyImage: New InputImage;
                Commune: New Communicate;
                Code: New Basic (Size = 4K, Source = ("Server.Cod"));
          )
```

When the template for the server module is instantiated, each component of the server module object is instantiated in turn. The mailbox template named **Communicate** and the basic template **InputImage** are defined outside of the module template for maintenance convenience. When the server template is instantiated, the names **Communicate** and **InputImage** are textually replaced by the mailbox and basic templates.

A particular object instantiation may refine a previously defined template, adding more detail about the object which is created from the template. For example,

```
Mailbox Communicate (MsgType = "Capability")

Module Server is
Construct (
              MyImage: New InputImage;
              Commune: New Communicate (Size = 10);
              . . .
              )
```

is also possible. Here, the template **Communicate** is defined outside the module template. Then the template is used within the module template, but with an additional parameter, the size. Note that, in TASK, it is not possible to define a template within a template; hence the **Communicate** template had to be defined outside the **Server** template.

Table 6-5: Syntax for TASK Parameters and Attributes

```
<Formal Parameters>  ::= <Formal Parameter>* | <Actual Attribute>*
<Formal Parameter> ::=
            {<Formal Parameter Name>,+ | <Iteration>} : <Simple Template>,
          | <Var Name>,+ : <Var Type>,

<Actual Attributes>  ::= <Actual Attribute>*
<Actual Attribute>   ::= {<Keyword Name> = {<Expr> | <Var Expr> | <Index>},}
                       | Source {[<Integer>, <Link Switches>]}#
                                  = <Source Parameter Value>,
                       | Rights = (<Quoted String>, <Quoted String>),
                              Currently, all rights specifications remain unused.
                       | Size = (<Integer>, <Integer>),
                       | <Special Attr>,

<Actual Parameters>  ::= {<Actual Parameter>* | <Actual Attribute>*}
<Actual Parameter>   ::= <Key Expr> = <Actual Expr>,

<Actual Expr>  ::= <Object Name> | <Iteration> | <Expr>
                 | <Var Expr> | <Access Expr> | <Index>

<Key Expr>  ::= <Formal Parameter Name> | <Access Expr> | <Iteration>

<Source Parameter Value>  ::= ({<Quoted String>},+)
<Link Switches>  ::= <Quoted String>
                        One or more switches (/D, /N, etc.) to be passed
                        to the C.mmp linker.

<Obj Common Att>  ::= Source | Rights | Size

<Obj Add Att>     ::= StackSize | Class | Preempt | Quantum
                    | ServiceLimit | ProcessId | MsgType
                        For a description of these attributes, see Appen-
                        dix C.

<Obj Special Att>  ::= Window | Stack | InitialCode
                     | PrivateMailbox | StackOwns | Alias | Present
```

We now adapt the previously defined server module to use a shared mailbox that is passed as an argument at the time of instantiation.

```
Basic InputImage (Size = 4K, Source = ("SeaScape.obj"))

Module Server (Commune: Mailbox (MsgType = "Capability")) is
Construct (
                MyImage: New InputImage;
                Code: New Basic (Size = 4K, Source = ("Server.Cod"));
            )
```

A name defined as a formal parameter to a complex template may be referred to in the body of the template, just as a name which is defined within the body of the template. Thus, **Commune** is a component of the server module; it can be named or manipulated just like any other component. The TASK compiler performs parameter type-checking. For example, in the most recent definition of the server module, **Commune** is a formal parameter restricted by a template. The actual parameter is thus required to be a capability mailbox—not a data mailbox. The template for **Commune** could have specified size attributes, although in this example it did not. Table 6-5 gives the syntax for TASK parameters.

Resource directives may be included in a complex template. We postpone discussion of them until later.

A.6. Modules, Functions and Processes

A STAROS *module* exports a set of *functions*. Thus, part of a module template consists of a specification of the functions defined by the module. A complete module template has the following form, as was shown in Table 6-3:

⟨Module Description⟩ ::= **Module** *⟨Module Name⟩* (*⟨Formal Parameters⟩*) **is**
⟨Construction Description⟩
⟨Function Description⟩ +
⟨Resource-Usage Directives⟩

where

⟨Function Description⟩ ::= **Function** *⟨Function Name⟩* (*⟨Formal Parameters⟩*) **is**
⟨Construction Description⟩
⟨Resource-Usage Directives⟩

Consider an expanded version of the server module template. This example defines a function called **WorkCycle**. The example supposes that a single process for the function **WorkCycle** will be created. This process is programmed to cycle, processing work requests sent to it via its private mailbox.

```
Module Server (InputImage: Basic, Commune: Mailbox) is
Construct (
        TheCommune: Process Server.WorkCycle();
        Code: New Basic(Size = 4K, Source = ("Server.Cod"));
        )
        Function WorkCycle is
        Construct (
                MyMailbox: Ref Commune(PrivateMailbox);
                MyStack: New Basic(Size = 4K, Stack);
                MyCode: Ref Code(InitialCode["Cycle"]);
                Scratch; New Basic (Size = 4K);
                )
        Directives()
Directives()
```

Example 6-1: TASK: An *Absent* Function

This example illustrates the use of *pathnames* (for example, **Server.WorkCycle**) to name functions within a module. Arbitrary paths of objects are created by concatenating component names ordered by their lexical nesting, with each pair of names separated by a dot ("."). For the present, the only legal pathname is one of the form *<Module Name>* . *<Function Name>*.

The **Process** construct results in the creation of a process to execute the function **WorkCycle**, which is defined a few lines below. This process will be created by invoking the *Process Creator* and not as the result of an ***Invoke*** instruction. *Any* **Process** construct causes a capability to be returned for the private mailbox of the new process; however, as in this case, the user can specify what mailbox object is to be used for the private mailbox. Also, a capability for the private mailbox will be placed in the slot of the module called **TheCommune**.

The last object which is created from the function template is a basic object called **Scratch**. The process object which is created by the **Process** construct from the function description contains capabilities for each of the four objects named in the function template. In general, the object

resulting from the instantiation of a *complex template* contains one capability for each component object. **New** causes the creation of an object by the object manager. **Ref** results in a ***Copy Capability*** operation which generates a new capability for the shared parameter object. In the BLISS code, the names given to object instances resolve to name slots in the capability portion of the complex object.

The **PrivateMailbox** attribute specifies that **Commune** will become the private mailbox of the new process. Note that **Commune** is a parameter to the module in this example. When the **PrivateMailbox** attribute occurs in the construction description of a particular function, it marks the object which it applies to as being the *private mailbox* of the function. Since each new process receives a unique private mailbox, and since the **Process** construct returns whatever object is the private mailbox of the process, the user needs to specify only when the default mailbox is to be replaced by another object (a module parameter in the example).

The fact that the mailbox is specified as a **Ref** to the object called **Commune** means that it is a reference to—that is, a capability for—**Commune**, which is the formal parameter to the module.[26] In other words, the capability which is passed to the module **Server** is to serve as the private mailbox for any process created from the **WorkCycle** function. The capabilities **Commune**, **MyMailbox**, and **TheCommune** are all copies of the same capability.

> Suppose a **Ref** for the **PrivateMailbox** had not been used. In other words, suppose that instead of "**Ref Commune**", we had written "**NewCom: New Mailbox (...)**". In this case, a new private mailbox would have been created indeed, if there were two **Process** constructs in the construction description for the module, two processes with two separate private mailboxes would have been produced.
>
> There is one more thing to note. By writing "**NewCom: New Mailbox (...)**", we gain the ability to specify any desired attributes for the new private mailbox that is to be created.
>
> By contrast, consider another modification of the example (below). In this example, the construction description does not specify an private mailbox. Consequently, when the process is created, a private mailbox with default size is returned in the capability slot **PMailbox**.

```
Module Server (InputImage: Basic) is
Construct (
        PMailbox: Process Server.WorkCycle();
        Code: New Basic(Size = 4K, Source = ("Server.Cod"));
```

---

[26] Formal parameters to one complex template (the module template) can also be referenced from inside a sub-template (the function template).

```
        )
        Function WorkCycle is
        Construct (
                MyStack: New Basic(Size = 4K, Stack);
                MyCode: Ref Code(InitialCode["Cycle"]);
                Scratch: New Basic (Size = 4K);
                )
        Directives()
Directives()
```

Special Attributes. The `PrivateMailbox` attribute is the first of the *special attributes* defined in TASK. These are attributes which are intimately related to characteristics of the STAROS system. Special attributes may appear in the same places as other attributes, such as `Size` and `Source` (which are defined in Table 6-5). `Stack` is another special attribute; it refers to the *process stack* of a STAROS process. The construction description for the `WorkCycle` function therefore says that a new 4K-byte basic object called `MyStack` should be created to serve as the process stack, each time a process is created from this function description. (The stack should always be a 4K-byte basic object, so that references to it do not need to be mapped through the Kmap).

Table 6-6: The Syntax of Special Attributes

<Special Attr>	::=	**Window** [*<Integer>*]#
		| **PrivateMailbox**
		| **InitialCode** [*<Quoted String>*]#
		| **Stack**
		| **StackOwns** [*<Object Name>*]
		| **Present** [*<Object Name>*]
		| **Alias** [" *<Function Name>* "]

The `InitialCode` attribute is also used in the example. It indicates that when the process is started, the entry point will be in the object `MyCode`, at the routine named `Cycle`. Note that `MyCode`, just like `MyMailbox`, is a `Ref` to an object defined in the enclosing module template. The effect is that each time a process is created from the function description, it is given a capability for the *same* code object; in other words, all processes created from the function description will share code. (If no `InitialCode` is specified for a function, TASK will generate a reference to the undefined symbol

$DefCall so that the linker will also generate a warning message.)

Certain objects should be in one of the fifteen window capability slots when a process begins execution. Objects with the attributes **InitialCode** and **Stack** must always be in a window; the TASK compiler takes care of this automatically. To cause other objects to be initially loaded into a window, they may be given the special attribute **Window**. In the following example, the basic object **Scratch** would be initially loaded into window 7:

```
Scratch: New Basic (Size= 40, Window[7])
```

The **Stack** and **InitialCode** objects will always be loaded into windows 0 and 1, respectively, regardless of whether they are given a **Window** attribute. In general, if no bracketed number follows the "**Window**", TASK will automatically select a window.

In STAROS programs, it is common practice to keep own variables above the stack,[27] but on the same page. If an object is given the attribute **StackOwns**, the stack object will be initialized from the named object. In this case, the function attribute block will indicate that the stack pointer should have the initial value *StackTop*, which should be a global name declared by the user. If **StackOwns** is not specified for a function, the initial stack pointer will be two less than the size of the **Stack**. (The **Stack** is always in window zero.)

Present/Absent* Functions.** When processes are created using the **Process** construct the question of whether the function is *absent* or *present* may not be important. However, if some function is intended to be the target of an ***Invoke instruction, the distinction is crucial (the difference between the to kinds of functions is discussed in [149]. When a function is present, ***Invoke***s directed to it are sent to the invocation mailbox which is named in the function attribute block. In TASK, a present function is defined by making the invocation mailbox, preceded by the word **Present**, a parameter to the function description:

---

[27] Since the stack grows downward, this means that owns are kept "at the base of the stack."

```
Module Server (InputImage: Basic, Commune: Mailbox) is
Construct (
        TheCommune: Process Server.WorkCycle();
        Code: New Basic(Size = 4K, Source = ("Server.Cod"));
        )
        Function WorkCycle (Present[Commune]) is
        Construct (
                MyStack: New Basic(Size = 4K, Stack);
                MyCode: Ref Code(InitialCode["Cycle"]);
                Scratch: New Basic (Size = 4K);
                )
        Directives()
Directives()
```

Example 6-2: TASK: A *Present* Function

In this example the object **Commune** is to be made the invocation mailbox for the *present* function **WorkCycle**. In this case, the invocation mailbox **Commune** happens to be a parameter to the module, although that is not necessary. If no **Present** attribute appears, the function is defined to be *absent*. Alternatively, if we had omitted the parameter **Commune** from the **Server** module, and written instead "**Present** [**TheCommune**]", the private mailbox of the process created when the task force was loaded would have become the invocation mailbox for the function.

If a function is *absent*, or if the task template does not specify a **Process** construct for a *present* function, a new process will be created if the function is invoked. Since this new process is created *after* the task force has been loaded, only the information specified in the function attribute block is available to initialize the process.

- The process stack will be as specified (**Stack**), but it will not be initialized (**StackOwns**).
- The object identified as **InitialCode** will be in the proper window.
- The private mailbox will be a unique object with the default size.
- Except for the **Stack** and **InitialCode**, no other objects will be placed in windows.
- No **New** objects will be created regardless of the function template, except for the stack.
- No **Ref** capabilities will be created regardless of the function template, except for one code page.

The *Alias* Attribute. The `Alias` attribute sets the alias function field of the function attribute block [149]. The effect of this is to cause the same process to perform two functions of some module. As a result, certain attributes of the alias function will be inherited from the function "aliased to". Hence, if the `Alias` attribute is specified, the `Stack` and `InitialCode` attributes should not be specified. For example:

```
Module Server (InputImage: Basic, Commune: Mailbox) is

        Function Func1 (Present[TheMBox]) is
        Construct (
                MyStack: New Basic(Size = 4K, Stack);
                MyCode: Ref Code(InitialCode["Cycle"]);
                Scratch: New Basic (Size = 4K);
                )

        Function Func2 (Present[TheMBox],Alias["MyMod.Func1"]) is
        Construct (
                Scratch: New Basic (Size = 4K);
                )
```

Example 6-3: TASK: Alias Function

The *Use* and *Name* Constructs. In all of the the `WorkCycle` examples thus far, the `Ref` construct has been used to denote the `Code` object in the server module. This is not sufficient if the programmer wishes to instantiate `WorkCycle` processes with their own copies of the code. In TASK, a process obtains a private copy of code or data objects through the `Use` construct:

```
Function WorkCycle is
  Construct ( . . .
              MyCode: Use Code;
              . . .)
```

In this case, the object specified by `Code` in the server module is *used*. This means that a copy will be made, if in the resource-usage directives the user has separately *localized* the object (see Section A.10). Otherwise, the `Code` object in the module will be used; it is `Ref`erenced.

The `Name` construct is provided to define elements with complex objects so that the elements may be named conveniently in BLISS code. TASK does nothing but allocate empty slots in the template. The BLISS program may use these slots to hold capabilities for dynamically created objects. A simple

example of the use of Name would be to reserve a slot in the *process name space*.

```
Function WorkCycle is
  Construct ( . . .
              CycleCode: Use Code;
              WorkSlot: Name;
              . . .
              )
```

If this name is to refer to a *window slot*, the programmer should write instead—

```
WorkSlot: Name(Window)
```

The *Reserve* Construct. For the purposes of assignment and placement, the Reserve statement may be part of a template's *⟨Construction Description⟩*. As noted before, all objects specified in templates are constructed before a task force can run. However, during execution new objects may be allocated, old ones deleted; processes may be spawned; and previously spawned processes may terminate. For such dynamically created and manipulated objects the Reserve construct is used. For example,

```
MyDynamicObject : Reserve Basic (Size = 4K);
```

reserves the name MyDynamicObject and associates it with a particular capability slot. All modules which make up the task force can use the name. The type and the actual parameters (e.g. Size) are effectively treated as comments; they indicate the programmer's intention to create a 4K-byte basic object in the slot, but there is no enforcement. At run time, the object which is actually created to occupy the slot may be quite different, and no warning or diagnostic will be issued.[28]

A.7. Task Forces

The root of any distributed application is a *task-force description*. At a minimum, a task-force description contains one module description. The form of a task-force description is similar to that of any complex object, except that it may have no parameters:

---

[28] The only difference between the Name and Reserve constructs is that Reserve causes the allocation of memory which can later be used to hold the object. However, since the *Memory Manager* does not have the ability to reserve space for later object allocation, the two constructs are at present equivalent.

TaskForce *<Complex Template Name>* **is**
<Construction Description>
<Resource-Usage Directives>

To illustrate a task force, let us assume that the module template named **Server** is defined as above and that there exists another module called **Coordinator** that requires two parameters, the image to be processed and a mailbox; and exports one function called **Control**. A process executing **Control** creates and passes work requests to the processes that result from invocations of the **WorkCycle** function defined in the server module. All server processes use the same **PrivateMailbox**, which is created in the task-force object, and then passed to the module template, and then to the function template. Example 6-4 is the definition of this task force.

```
TaskForce ImageProcessor is
Construct (
        TheImage: New Basic (Size = 4K, Source = (Image));
        PMailbox: New Mailbox (MsgType = "Capability", Size = 25);
        CoordMod: New Coordinator (InputImage = TheImage,
                                   Commune = PMailbox);
        ServerMod: New Server (InputImage = TheImage,
                               Commune = PMailbox);
        )
Directives()

Module Server (InputImage: Basic, Commune: Mailbox) is
Construct (
        ...
        ServerPM1: Process Server.WorkCycle ();
        ServerPM2: Process Server.WorkCycle ();
        )
        Function WorkCycle is
        Construct (
                CommBox: Ref Commune (PrivateMailbox);
                ...
                )
        Directives()
Directives()

Module Coordinator (InputImage: Basic, Commune: Mailbox) is
Construct (
        ...
        ManagerPM: Process Coordinator.Control (CommBox = Commune);
        )
        Function Control is
        Construct (
                CommBox: Ref Commune (PrivateMailbox);
                ...
```

```
                    )
            Directives()
Directives()
```

Example 6-4: A Simple Task Force

When a task force is instantiated using the **ImageProcessor** template, a StarOS task-force object is created. All of its components, including the two module objects, are created and initialized. Instantiation of the module templates will result in the creation of all objects specified in the modules' construction descriptions, thereby creating the three processes. This "chain-reaction" instantiation of objects makes the construction of a task force easier for the programmer by taking away the need to re-code all of the individual actions involved in the construction of the task force.

It is the loader which creates the task-force object. The task-force object is known to the loader as the *loader library*. It places in this object a capability for each object instantiated within the task-force template. After the loader finishes creating the task-force object, and anything else which it causes via the chain-reaction, it gives the module capabilities in the task-force object to the user interface. This enables the user to ***invoke*** functions of these modules from command level.

> Task creates one *template file* for each module and one template file for the task-force object. These files consist of initialization *information* for the module and *function attribute blocks*, as well as directives to the *loader* to create certain objects. Each module's template file is assembled, then linked into the corresponding module (.MOD) file by the C.mmp linker before being sent to Cm*. For each task force, a separate file is output by the C.mmp linker (.TF file). Figure 6-2 oversimplified the process somewhat, since it showed the .TEM files being passed directly to the *loader*. In reality, though they consist solely of loader information, they are combined with the corresponding .MOD or .TF files before being passed to the *loader*.

Scope Rules. In a Task program, all template names are global, and within one Task program they must be unique in their first five characters. Function names are nested, as in Algol: if a particular name occurs twice, an instance of the name refers to the one in the smallest enclosing complex template: complex-basic, module or task-force description. Object names are known only within the complex template in which they are defined.

It is now evident why the private mailbox was passed to the module as a parameter in the last

example. The scope of the name PMailbox is limited to the task-force template, so it cannot be referred to from inside the module templates. (By contrast, objects defined in a module template can be referred to from within any associated function template, since the module description logically includes the function description; see Table 6-3.) Instead of using the Ref construct, we could have declared CommBox as a parameter to the function template. Then we might have written

```
Function Control (CommBox: Mailbox (PrivateMailbox)) is ...
```

The two constructions are semantically equivalent, but parameterized function templates cause the compiler to generate less space-efficient loader instructions (.TEM files), which for large task forces may cause the loader's instruction buffer to overflow.

A.8. Iteration

In the complex templates of the previous section, each component is explicitly named. However, to specify distributed software where the number of components varies with the size of the input data involved or the number of processors available, specification of each explicit component is inconvenient.

In this section we introduce *iteration* for the specification of multiple components that differ only in minor ways. Iteration can be used to express the *partitioning* of data objects into multiple objects, and for specifying parameterized *replication* of objects.

To illustrate replication, we adapt the ImageProcessor example to use a variable number of server processes. It is necessary to modify only the Server module description:

```
Module Server(InputImage: Basic,
              Commune: Mailbox, number: Integer) is
Construct (
                . . .
                (i=1..number Max 12) ServerPM[i]:
                    Process ServerMod.WorkCycle (CommBox = Commune);
                )
```

The iterative syntax above will cause the creation of number processes, each to executing the WorkCycle function. Number potentially varies each time the task force is instantiated, but may not exceed the value 12.

To illustrate the use of iteration for the purposes of partitioning data, we present another example: StarOS restricts object data parts to 4K bytes. To alleviate this restriction, the Task iteration facilities may be used to partition files larger than 4 Kbytes into multiple basic objects. For example, inside a **Construct** clause, one may write:

```
(i=0..2) MyCode[i]: New Basic (Size = 4K,
                               Source = ("Codfil.obj"));
```

Three code objects will be built from the file named `Codfil.obj`. `MyCode[0]` contains the first 4K bytes of code; `MyCode[1]` contains the second 4K bytes, and so forth.

Table 6-7: Syntax for Iterations

⟨Iteration⟩ ::= (*⟨IterName⟩* = *⟨Low Limit⟩* .. *⟨High Limit⟩*) *⟨Access Expr⟩*

⟨Access Expr⟩ ::= *⟨Iterated Name⟩* [*⟨Index⟩*]

⟨Index⟩ ::= *⟨IterName⟩* {**Mod** *⟨Var Expr⟩*}$^{\#}$
⟨Low Limit⟩ ::= *⟨Integer⟩*
⟨High Limit⟩ ::= *⟨Integer⟩*
| * **Max** *⟨Integer⟩*
| *⟨Integer⟩* **Max** *⟨Integer⟩*
| *⟨Variable⟩*

⟨IterName⟩ ::= *⟨Simple Name⟩*
⟨Iterated Name⟩ ::= *⟨Comp Name⟩*
| *⟨Formal Parameter Name⟩*
| *⟨Actual Parameter Name⟩*

⟨Expanded Iteration⟩ ::= *⟨Comp Name⟩*,$^{+}$
Expanded iteration and arithmetic expressions are not implemented.

If the task-force author does not know how many objects will result from partitioning a source file, the symbol "*" is used to indicate "as many as necessary to exhaust the file". Rewriting the above example using the "*" notation, we have:

```
(i=0..* Max 5) MyCode[i]: New Basic (Size = 4K,
                                     Source = ("Codfil.obj"));
```

The maximum range of the iteration index must be known at compile time. Thus, no more than 6 **MyCode** objects will be created, regardless of the size of the file `Codfil.obj`. Furthermore, the last **MyCode** object will be 4K bytes in length, regardless of whether it is completely filled.

Once a name (such as **MyCode**) is associated with an iteration, subscript notation may be used to select an object in the range [*<Low Limit>* . . *<High Limit>*]. In the previous example, we can refer to `MyCode[1]`, which is the basic object containing the second 4K bytes of `Codfil.obj`. *<High Limit>* can be an identifier name or an integer. Table 6-7 defines how iteration may be used in construction clauses.

A.9. Parameters

There are two classes of parameters to templates:

- *Predefined parameters* in object templates, such as the size of the object, and the message type (for a mailbox). Such parameters are called *attributes* and a complete list of them appears in Appendix C.
- *User-defined parameters*, such as the parameter **Commune** in the module **Server** (Example 6-4).

A specific set of predefined parameters (*attributes*) is associated with each type of object. When a template is used for object construction, those predefined parameters that are not specified are given default values by the compiler. Predefined parameters include **Size**, **Source**, and **MsgType**:

```
Basic InputImage (Size = 4K, Source = ("Image1.obj", "Image2.obj"))

Mailbox Communicate (MsgType = "Capability", Size = 40)
```

When integers are to be specified as actual parameters, integer constants, simple names and arbitrary expressions[29] are allowed.

Predefined parameters may also require objects as parameters. In the **WorkCycle** function, for instance, the **PrivateMailbox** is a predefined object parameter.

---

[29] Not implemented.

When a formal parameter is declared, a type must be associated with it, and it may additionally be constrained by a template. When the template is used for constructing an object and actual arguments are associated with these parameters, the compiler prints a warning. In any case, the compiler and the loader will make all attempts to proceed with object construction. Non-constant arguments may be expressions, which evaluate to *<string>* or *<integer>* values. Variable arguments are not available at run time.

In the following example, the module **Server** has two formal parameters—**InputImage** and **Commune**:

```
Module Server (InputImage: Basic,
          Commune: Mailbox (MsgType = "Capability", Size = 10)) is
Construct (
                Code: New Basic (Size = 4K, Source = ("Server.Cod"));
           )
```

Here, the **Commune** parameter is constrained to be a capability mailbox with room for at most 10 capabilities. Both parameters are objects.

An example of a variable parameter is—

```
Module Image (Image: String) is
Construct (
                InputImage: New Basic (Size = 4K, Source = (Image));
                . . .
                )
```

It is possible to use iteration when expressing formal and actual parameters,[30] as illustrated here:

```
TaskForce ImageProcessor is
Construct   (
                (i=1..* max 20) TheImage[i]: New Basic (Size = 4K,
                                        Source = ("Image.obj"));
                CoordMod: New Coordinator (InputImage =
                                        (i=1..* max 20) TheImage[i]);
                        )
Module Coordinator ((i=1..n max 20) InputImage[i]: Basic(Size=4K)) is
Construct ( . . . )
```

Here an instance of a **Coordinator** module may have up to 20 parameters, called **InputImage[i]**,

---

[30] Not implemented.

passed to it. A maximum value must be supplied with the parameter, thereby constraining the iteration and limiting the number of `InputImage[i]` objects within the *<Construction Description>*. In this example, each `InputImage[i]` parameter must be of size 4K bytes.

When referred to in the construction description, the vector name can be referenced to denote the entire vector. Alternatively, its elements may be referenced individually using an indexed vector name. Here is another example of parameter iteration:

```
Module Coordinator ((i=1..n max 20) InputImage[i]: Basic) is
Construct (
                    . . .
                    TestElement: Ref InputImage[0];
                    )
```

Several formal parameters which have the same type or template may appear in a list, separated by commas.[31] The type need only be specified once, following the last parameter of this type. This is illustrated by the following:

```
Module Server (InputImage1, InputImage2: Basic) is . . .
```

6.0.1. The *Source* Attribute

The `Source` attribute serves two functions. It indicates what files are to provide the data necessary to initialize the data part of an object; and, if it is necessary to link-edit the the data files, the relocation address to be used by the C.mmp linker (the link editor used by STAROS). In general, all compiler output (relocatable object file, `.OBJ`) must be link-edited for two reasons: first it is necessary to calculate addresses of items such as routines and global variables that may be part of the data; second, code and linked data structures may include pointers that depend on where these structures are located in the address space.

The full syntax of the `Source` parameter (from Table 6-5) is:

---

[31] Not implemented.

<Source Attribute> ::= **Source** {[*<Integer>*, *<Link Switches>*]}$^{\#}$
= *<Source Parameter Value>*,

,*<Source Parameter Value>* ::= ({*<Quoted String>*},$^{+}$)
<Link Switches> ::= *<Quoted String>*

A *<Source Parameter Value>* is used to specify the files that contain the data to be used to initialize the object. In previous examples, the specification "**Source = ("Image1.obj")**" has been seen. This specification indicates that the data in the file **Image1.obj**, after processing by the linker, is to be used to initialize some newly created object. Since all files used in this manner will be processed by the linker, these files must be in the so-called *obj format*. Such files are most conveniently generated using some language translator.

In general, **.OBJ** files generated by BLISS are partitioned into named **csects**, each *csect* representing a different kind of information. Accordingly, file names in a **Source** clause may be qualified to select particular *csects*. A *<Source Parameter Value>* of "**Source = ("CodFil.obj(CG)")**", for example, means that (only) the code and globals from the file **CodFil.obj** should be placed in the object. The characters in parentheses select a particular type of *csect*: **C** selects code, **G** selects globals, **O** selects *owns*, and **P** selects *plits*—literals generated from the BLISS program. A *<Source Parameter Value>* of "**Source = ("CodFil.obj </P MyPlits>")**" means that the *csect* of type *plits* called "**MyPlits**" should be extracted from the file **CodFil.obj**. Lastly, "**Source = ("CodFil.obj </C Front>")**" instructs the linker to extract the *code csect* called "**Front**".

An object containing code or data that must be relocated often has the **Window** attribute specified. In such cases the programmer need not specify the relocation address for the object since this is implied by the window assignment, and TASK will automatically generate the proper instructions for the linker. Circumstances can arise where the programmer wants to relocate several objects to the same address or does not want the loader automatically to place some object in a window. Then there must be a way to provide the relocation value in an independent fashion, and this is the purpose of the *<Integer>* value. For example, "**MyObject(Source[4, ...] ...**" indicates that the contents

of MyObject should be relocated to page 4 (address #040000)."

By way of illustration, suppose that executable code is to be overlaid because the address space of the process is too small to contain all of the code. In this case, two or more objects will be initialized from different code fragments which have to be relocated to the same address. The programmer cannot use the Window attribute in the body of a function template, for that would cause capabilities for both fragments to be loaded into the same window at initialization time, with one overwriting the other.

The Source attribute is designed for this situation. Like the Window attribute, it causes code within an object to be *relocated* relative to some window, but unlike Window it does not cause a capability for the object to be *loaded* into a window. In our illustration, one of the two code objects would probably be loaded into the window at initialization time through use of the Window attribute; the other code object would be given a Source attribute *with the same window number as the first object.* Then, when the program was ready to use the second object, it would use a ***Load Window*** instruction to load a capability for it into the window, making it addressable in place of the first code object. A somewhat more elaborate example, which uses four overlaid code pages instead of two, follows.

```
Module Overlay is
Construct (
        Code0:  New Basic (Source[7, " "] = ("Code0.obj"));
        Code1:  New Basic (Source[7, " "] = ("Code1.obj"));
        Code2:  New Basic (Source[7, " "] = ("Code2.obj"));
        Code3:  New Basic (Source[7, " "] = ("Code3.obj"));
              )
        Function DoIt is
        Construct (
                OverlaidPage: Ref Code0(Window[7])
                      )
```

When the process is created, the object with Code0 will be in window 7. To address the other objects one must code:

```
LoadWindow(OverlaidPage, Code1);  ! or Code2, Code3, etc.
```

In this example, the second argument of the Source attribute is the *<Link Switches>* character

string, consisting of exactly one blank. The current release of the TASK compiler does not allow the string to be omitted. Nor may the string be null; at least one blank is required. Note that if the code objects were to be relocated to window 0, the brackets and the text inside could have been omitted completely; relocation relative to window 0 is the same as no relocation at all:

```
Code0:  New Basic (Source = ("Code0.obj"));
```

The *<Link Switches>* allow the programmer to specify switches for the C.mmp linker. This parameter is copied without change to the .LMD file generated by TASK. The switches of primary interest to the user are:

/D Indicates that this is a "dummy" page. The indicated *csects* will be link-edited so that global symbols will be bound, but no page will be output. Typically /D is specified for the stack page since each new process will receive a new, uninitialized page regardless.

/O Forces the linker to link the specified *csects* in the exact order given. Otherwise, the linker may change the order of the *csects* to reduce disk usage.

/N and /S These switches label pages containing name and symbol-table *csects* used by the debugger [149].

6.0.2. BLISS Names Generated by TASK

TASK generates .DFS files which can be required by a BLISS program, making the object names declared in the TASK program available to the BLISS code. In the example

```
Module TwoLevel is
Construct (
        . . .
        ComplexBasic Compound is
        Construct (
                First: New Basic(Size = 4K);
                . . .
                Scratch: New Basic(Size = 4K, Window[7]);
                . . .
                )
        Directives()
        )
Directives()
```

the name Scratch refers to the same object (more precisely, to the same capability slot) in the BLISS program as it does in the TASK program. However, a capability slot is named by a *(primary index,*

secondary index) pair [149]. The **.DFS** file includes two bindings for each slot named in the TASK program; if *name* appears in the TASK program, then in the **.DFS** file, *name* will be bound to the capability index, while **$name** will be bound to the secondary index. For example, if **Scratch** is found in slot **(5,14)**, then in the BLISS file, **Scratch = (5, 14)**, while **$Scratch = 14**.

There is another instance in which TASK generates an extra name. If the **Window** attribute is specified as in the above example, then TASK generates an extra name to refer to the base address of the page through which **Scratch** will be referenced, i.e.,

```
ScratchPage = 70000;
```

A.10. Resource-Usage Directives

Resource-usage directives allow the programmer to control where task-force components are placed. The objective is to allocate memory and processor resources to the task force. We call the complete set of allocations the *map* of the task force to the architecture. This *map* has two parts. First, there is *placement*: every object must be represented somewhere in physical memory. Second, there is *assignment*: each process must be assigned to execute on some processor in the system.[32] Sometimes, users need to make explicit resource-allocation decisions. TASK allows users to specify none, parts of, or all of the task-force mapping. In addition, or as a preferable alternative, users may express *constraints* or *preferences* on how resources may be allocated to instantiated objects. A constraint effectively eliminates some maps, but does not necessarily determine a specific map, or even a partial map. A preference eliminates these same maps only if the compiler chooses to honor the preference. Constraints and preferences are used by the TASK compiler and the loader to create a map specification for the task force, given the available physical resources.

---

[32] In STAROS, processes are assigned to run queues instead of processors. A run queue can be serviced by more than one processor. TASK assumes the default run-queue mapping, that is, the compiler assumes that each run queue is serviced by exactly one processor. Then, run-queue names and processor names are identical. If the user has set up a version of STAROS with a different run-queue mapping, TASK placement and assignment cannot be relied upon.

6.0.3. Proximity Relations

Constraints are expressed in two forms: as *proximity relations* between pairs of software components, and as relations between software components and physical resources.

For example, to enhance reliability, a task force might require that each of three replicated processes execute in a different Cm* cluster. In this example, precisely which processors are to be used is unimportant. However, each pair of processes should/must not execute closer than **DifferentCluster**. As a second example, to maximize performance perhaps some process should always execute with local code. In this case, the proximity relation between the processes and their code should be **SameCm**. In both examples, a proximity relation is expressed between two software components without concern about the physical computer modules involved.

TASK provides six *degrees of proximity* that mirror the range of performance characteristics exhibited by the Cm* hardware.[33] The TASK proximity relations are:

- **SameCm**—Should/must be in the same computer module.
- **SameCluster**—Should/must be in the same cluster and can be in the same computer module.
- **NearCm**—Should/must be in different computer modules and should/must be in the same cluster.
- **DifferentCm**—Should/must be in different computer modules and can be in different clusters.
- **DifferentCluster**—Should be in different clusters.
- **NoCare**—Can be anywhere. This is the proximity degree that is given by default to all software components that are not related with explicit directives.

A *resource-usage directive* specifies a proximity relation between pairs of objects. The syntax, given in Table 6-8 permits users to list sets of objects for which the relation of interest is to hold. For

---

[33] Were we dealing with a different architecture, such as a network including a store-and-forward switch, we would alter the proximity relations slightly. They would express the "distance" between two resources, measured in the expected number of message stores required to transmit a message.

Table 6-8: Syntax for Resource-Usage Directives

```
<Resource-Usage Directives> ::= Directives (<Selection>;*<Directive>;*)

<Selection>    ::= {<Hardware Set Name> |<Var Name>} : <Selection Expr>
<Directive>    ::= <Iteration># <Proximity Degree>
                          ({<Iteration># <Object Path Name>)},+)
                 |<Iteration># Same (<Hardware Set Name> [<Index>]#,
                          {<Iteration># <Object Path Name>},+)
                   In the first implementation no paths will be used.

<Proximity Degree> ::= <Integer> | SameCm | SameCluster | NearCm
                     | DifferentCm | DifferentCluster | NoCare
<Selection Expr>   ::= <Set Expr> {where <Attr Expr>}#
                     | AnyOf <Set Expr>
                     | NumberOf <Set Expr>

<Set Expr>     ::= <Set Name> | (<Set Name> <Opr> <Set Name>)
<Set Name>     ::= <Predefined Set Name> | <Hardware Set Name>
<Attr Expr>    ::= <Hard Att> <RelOpr>
                 | (<Hard Att><RelOpr><Hard Att Value><Opr><Hard Att>
                    <RelOpr><Hard Att Value>)#

<Opr>          ::= and | or
<RelOpr>       ::= < | > | = | <= | >=

<Hard Att Value> ::= <Integer> | True | False
<Hard Att>     ::= MaxSize | ActualSize | Reliability
                 | Speed | NumEther | NumLines | NumDisks
                 | NumDALinks | NumCms | NumClusters
                 | HasDALink | HasDisk | HasEther | HasLine
                 | MaxMPSize | ActualMPSize MPReliability
                 | PCReliability | PCSpeed

<Predefined Set Name> ::= Cm | CmStar
                        | Cluster0 | Cluster1 | ... | Cluster4
                        | Cm[0,0] | Cm[0,1] | ... | Cm[0,14]
                        | Cm[1,0] | Cm[1,1] | ... | Cm[1,14]
                        | Cm[2,0] | Cm[2,1] | ... | Cm[2,14]
                        | Cm[3,0] | Cm[3,1] | ... | Cm[3,14]
                        | Cm[4,0] | Cm[4,1] | ... | Cm[4,14]

<Hardware Set Name> ::= <Simple Name>
```

instance, in the WorkCycle directives on page 159, the author could state that WorkCycle processes are to have SameCm proximity to both their process stack, called MyStack, and the Scratch object used for writing intermediate results:

```
SameCm (MyStack, Scratch);
```

In addition, the author of the Server module template may prefer that the individual server processes all execute on different processors so that they do not compete for processor cycles. This is expressed using the iteration construct:

```
NearCm ((i=1..n) ServerPM[i]);
```

which is equivalent to stating that ServerPM[1], . . . , ServerPM[n] should be NearCm.[34] If two objects are to be SameCm, the TASK compiler understands that it is a preference concerning object placement that both objects must be placed into the Same computer module. However, the compiler will decide which particular Cm the objects are to be placed into.

The SameCm proximity relation implies a new "same-processor" relation that is transitive. For example:

```
SameCm (MyStack, Scratch);
SameCm (Scratch, MyCode);
```

requires that MyStack and Scratch, Scratch and MyCode, and consequently MyStack and MyCode are to be place in the same computer module's memory. Such implied relations can induce conflicts among proximity relations. A conflict occurs, for instance, if we add the directive

```
NearCm (MyStack, MyCode);
```

to the directives in the example above. The conflict is resolved by giving same preference over different. Thus, in the example MyStack and MyCode would remain SameCm, and in the same processor. To avoid conflicts of the form

```
SameCm (Code, Cm1);
SameCm (Code, Cm2);
```

each component of a template (e.g. Code) can be named in exactly one SameCm or SameCluster

---

[34] Note that for resource directives the private mailbox returned as result of process creation is used as a unique name to denote the created process. The directive does not state anything about placement of the private mailbox.

directive.

The NearCm directive implies a "same cluster" relation which is also transitive. For instance, the two directives

```
NearCm ((i=1..9) ServerPM[i]);
NearCm (Scratch, ServerPM[1]);
```

require that each server process must be NearCm to all other server processes, and also that the object called Scratch be NearCm to the first server process. All servers and the object Scratch will be placed in the same cluster.

The directive

```
DifferentCm ((i=1..50) ServerPM[i]);
```

requires that 50 server processes be assigned so that each process executes on a distinct computer module, regardless of the clusters in which the modules are located. Note that the proximity degree NearCm would have been too constraining in this case, since there are no clusters in Cm* with more than 14 computer modules.

Restrictions on Directives. We have already noted that each object can be named in at most one Same directive. There are a few more restrictions which apply to directives:

- Directives cannot name parameters (to the complex template in which they appear), and iteration bounds used in directives must be integers. Thus, to express directives for iterated components with variable upper bounds, the maximum iteration value should be used.
- Variables from the configuration table (see Chapter 4) cannot appear within templates; e.g. "(i=0..NumCms)" is invalid.

6.0.4. The Use of Directives for Optimization

So far, we have used directives to express preferences on resource allocation. The TASK compiler, however, uses these preferences and other information from the TASK specification to derive "good" placement and assignment decisions. To help the compiler improve its decisions, users can specify additional information about their task forces' expected run-time behavior. These are called *proximity*

estimates. Estimates are expressed as integer-valued proximity degrees (in the range of 1 - 100). High proximity between components is implied by large integers, whereas low proximity is encoded by small integer values. In fact, we associate encodings with the keywords introduced above:

- **SameCm**—corresponds to **90**,
- **SameCluster**—corresponds to the range **31 - 89**,
- **NearCm**—corresponds to **30**,
- **DifferentCm**—corresponds to the range **11 - 29**,
- **DifferentCluster**—corresponds to **10**, and
- **NoCare**—corresponds to the range **1 - 9**.

This encoding corresponds to the performance hierarchy in memory accesses within Cm*. The integer-valued proximity degree is interpreted as a *finer-grained estimate* of the need for two objects to be "close" to each other. No keyword covers the range 90 to 100; values in this range are used to express a stronger preference than even the **SameCm** keyword for placing objects in the same Cm. As an example, **88** is interpreted as **NearCm** and also informs the compiler's optimization phase that it is extremely important for the objects to be located in the same cluster. Thus,

```
SameCm (MyStack, Scratch);
```

and

```
100 (MyStack, Scratch);
```

would be identical with respect to the mapping constraints that have to be fulfilled by the compiler. For optimization, however, the values **90** or **100** would be used.

The integer encoding has another use in TASK. It is used to resolve conflicts between directives. The compiler simply uses the integer encodings of proximity degrees as precedence values. Again, a choice is given to the user whether or not constraint or preference semantics should be attached to proximity estimates.

6.0.5. Using TASK With Distributed Hardware

All of the resource-usage directives we have encountered thus far have involved only software objects. A surprising number of mapping constraints can be directly expressed without mentioning explicit architectural resources. In fact, it is rarely appropriate for the task-force author to single out

one particular computer module for use in a proximity relation. Occasionally, one might be identified by name in special cases, perhaps when an engineer wishes to execute diagnostics on a particular computer module. In general, the task-force author will specify architectural resources by *attribute*, not by unique name. The attributes serve to select a subset of all resources. For the purposes of the constraints, the resources in the selected set are interchangeable. For example, a process with large memory demands might be constrained to execute on a processor with more than 64K bytes of local memory. If a Cm* cluster includes several computer modules with at least 64K bytes of memory, then this constraint would establish that one of those computer modules should be chosen.

For the purposes of referring to architectural resources within proximity expressions, we can define the hardware architecture to be an "architectural" task force, using TASK syntax which is much like the syntax for TASK's software templates. The TASK language can also define and ascribe user-specified names to sets of physical components selected by their attributes (a full description of how this is done will be given in the next section). These names could then be used in proximity expressions. In such a description, **DiskCm**, for example, might be the name of a computer module with a disk directly attached. Consider a software task force designed to execute on a single cluster of computer modules. Assume that the task force includes a disk-management process named by its private mailbox (**DiskManagerPM**). The directive

```
Same (DiskCm, DiskManagerPM);
```

specifies part of the mapping between software and architectural task forces so that the disk-management process must execute on a computer module with a disk. (In a **Same** directive, the hardware resource is specified first.)

The implementation of TASK does not allow the language-driven construction of arbitrary architectural task forces. Instead, we use a pre-constructed architectural task force that describes the Cm* architecture, using predefined *hardware component names*, predefined *hardware set names*, and predefined *hardware attributes*. Users can then refer to these names in directives, in the same manner that software components are named. The TASK compiler uses an internal table, called a

configuration descriptions, that describes the current Cm* hardware configuration. This table can be updated just before a compiler run, by using configuration information maintained by StarOS.[35] The standard configuration description is in file `ARCHIT.STD[x335cm01]`.

In configuration descriptions, we encode the following information (*hardware attributes*). For each computer module is recorded—

`MaxMpSize`	`ActualMpSize`	`MpReliability`	`PcSpeed`	`PcReliability`
`HasEther`	`HasLine`	`HasDALink`	`HasDisk`	cluster number

> `MaxMpSize` is size of a Cm's memory; `ActualMpSize` is the total memory size, minus the space occupied by the operating system. `MpReliability` and `PcReliability` will be used to encode the observed reliability of the Cm's memory and processor, respectively, in an integer value similar to the proximity degrees. Similarly, `PcSpeed` will encode of the speed of a processor; speed variations of 20% or so between processors are not uncommon. `HasEther`, etc. are boolean values indicating whether the Cm possesses the various pieces of hardware. The speed and reliability attributes are not yet implemented.

For each cluster, a list of the Cm's in the cluster and a list of *cumulative attributes* is given. The cumulative attributes are—

`MaxSize`	`ActualSize`	`Reliability`	`Speed`	`NumCms`
`NumEther`	`NumLines`	`NumDALinks`	`NumDisks`	

> All of these values except `NumCms` are merely the sum of the corresponding values for all the Cm's in the cluster. For example, `ActualSize` is the sum of the `ActualMpSizes`.

For Cm*, a list of clusters (`CmStar`) and a list of Cm's (`Cm`) are recorded. In addition, `NumClusters` is available as an attribute, so that it can be used, say, as a limit in an iteration statement.

We distinguish three different predefined *hardware set names*:

- `CmStar`—to name the set of clusters in Cm*,
- `Cm`—to name all computer modules in the Cm* system, and
- `Cluster0`, ... , `Cluster4`—to name each of the five clusters.

The predefined *hardware component names* are `Cm[0,0]`, ... , `Cm[4,14]`, which name each computer module in each of the clusters of Cm*. The first index denotes the cluster, and the second

---

[35] StarOS maintains the status information on Cm*. There is not yet any way to ship this file back to CMU-10A, where the Task compiler runs.

index the computer module within the cluster.

6.0.6. The Selection of Resource Sets

On page 182, the name **DiskCm** was introduced to refer to any Cm with a disk attached in one particular cluster, say, cluster 1. **DiskCm** is not a predefined hardware name; rather it is defined by a TASK *selection statement*.

A selection statement selects a member of a hardware set. Beginning with the predefined hardware sets, additional hardware sets can be specified using the **Where** construct. As an example, consider the set named **Cm64D**. Let **Cm64D** denote the set of all Cm's in **Cluster1** with more than 64K bytes of memory and an attached disk. To achieve this selection, we use[36]—

```
Cm64D: Cluster1 where (HasDisk = True and MpSize >= 64K);
```

To select an element of this set, we write—

```
DiskCm: AnyOf Cm64D;
```

In the latter selection, the **where** construct was not used. That implies that any one of the elements of the set **Cm64D** may be selected.

Thus the statement

```
Same (DiskCm, DiskManagerPM);
```

means that the disk-management process can be assigned to any one of the computer modules in cluster 1 with more than 64K bytes of memory and an attached disk.

There are three ways new sets of components can be derived from the predefined sets of hardware components:

- *Selection by name.* A single element (a singleton set) or a set may be named. For example, the name **Cluster1** names one cluster in Cm* (but not its components) and **Cm[1,1]** names Cm 1 in cluster 1.
- *Selection by arbitrary choice.* **AnyOf** ⟨*Set*⟩ names a single element of ⟨*Set*⟩.

---

[36] This notation has been inspired by the languages for relational [illegible]

ample, AnyOf Cluster1 names some one component of Cluster1, in this case a computer module. The user cannot rely upon a particular component being selected; an arbitrary one is chosen. AnyOf CmStar names one arbitrary cluster in the system, and AnyOf Cm names one arbitrary Cm in the system.

- *Selection by attribute*. A set may be formed using attributes. For example,

```
Cluster1 where MpSize = 64K;
```

selects the modules in Cluster1 with 64K bytes of memory.

```
AnyOf Cm where HasDisk = True;
```

selects any one computer module in any cluster that has a disk attached. Any of the predefined attributes listed in the configuration tables can be used in a selection based on attributes. When attribute values are associated with attribute keywords, a number of different *relational operators* can be employed. For example, the directive:

```
Cluster1 where MpSize >= 64K;
```

is a legal directive selecting all computer modules in Cluster1 with at least 64K bytes of memory. The available relational operators are "<", ">", "<=", ">=", and "=".

The NumberOf construct (not implemented),

```
NumberOf <Set>
```

denotes the number of elements in *<Set>*. For example,

```
NumberOf Cm64D
```

returns the number of Cm's which have 64K bytes of memory and a disk attached.

All sets have an attribute that encodes the number of Cm's in the set. For example, if we desired to select "large" clusters, we might write:

```
CmStar where NumCms >= 8;
```

This selects the set of clusters which have eight or more components. Note that whenever the value of an attribute is an integer, any relational operator may be used. Also note that the attribute keywords use with sets (the cumulative attributes) are distinct from those used with hardware components (the hardware attributes).

More than one attribute can appear after a where construct; if so, the list of attributes must be enclosed in parentheses. If multiple attributes are separated by the and operator, the selected set will

include the components that possess both attributes. This was illustrated by the set Cm64D defined above. An or construct is available, too. To express that a selected Cm should have either a disk or an Ethernet link, write

```
AnyOf Cm where (HasDisk = True or HasEther = True)
```

(The keyword False could also have been used.) Where clauses may not be nested. If the effect of nested clauses is desired, more than one selection statement must be employed. For example,

```
DskCluster: AnyOf CmStar where NumDisks >= 1;
```

selects a certain cluster, called DskCluster. Then,

```
AnyOf DskCluster where (HasDisk = True and MpSize >= 128K);
```

selects a particular Cm from DskCluster with a disk and more than 128K bytes of memory.

The AnyOf construct is always directly followed by a *set expression*. This can either be the name of a single set, for example, CmStar, or it can be an expression involving intersections, unions, or negations of multiple sets. For example,

```
AnyOf (Cluster1 or Cluster2);
```

expresses that any one of the components of Cluster1 or Cluster2 should be selected, and

```
AnyOf (Cluster1 and CmStar);
```

expresses that any one of the components of Cluster1 that is also a component of CmStar may be selected. Set or attribute expressions cannot contain multiple parentheses. There are no precedence rules between operators that combine sets or attributes; set or attribute expressions are evaluated left to right.

The syntax of TASK language constructs has been simplified, without any loss of generality, by forcing all selections to be made explicitly in the selection part of the Directives section. Thus, the directive:

```
Same (AnyOf Cluster1, Stack);
```

is not a legal directive. Instead,

```
StackCm: AnyOf Cluster1;
```

must appear as a selection, and

```
Same (StackCm, Stack);
```

must appear as a directive.

A final point needs to be made about set selection in TASK. If software components are related with Same to components of two different but partially intersecting hardware sets, then conflicts may occur. For example:

```
Same (A,a);
Same (B,b);
```

where a and b denote software components, and A and B are the selected hardware components, the statement

```
NearCm (a,b);
```

may lead to a conflict. The TASK compiler recognizes such conflicts and tries to select hardware components to avoid them. However, if A and B are singleton sets that intersect, an error is reported. In that case, TASK picks an arbitrary hardware component for either a or b. In this case, TASK issues a warning message.

6.0.7. An Extended Example with Directives

Consider a task-force author who wishes to use one cluster. The task force involves a variable number of processes:

- DiskManagerPM—which manages a disk dedicated to this task force,
- BossPM—a coordinating process that sends work requests to the server processes, and
- ServerPM[1]—the server processes.

Although there may be a maximum of six server processes, the actual number varies with the number of processors available.

To improve performance, the disk-manager process should execute on the computer module directly attached to the dedicated disk; the servers should execute on separate processors. We assume the "boss" does little processing in comparison to the servers, and hence does not need a private processor. In the selection part of the directives of the task force definition, we first select a

cluster that has at least 2 processors and one disk. Then we select the Cm with the disk. Because the selected resource set and the Cm with the disk have to be referred to again, we assign the names **MyCluster** and **DiskCm** to them.

```
MyCluster : AnyOf CmStar where (NumCms >= 2 and NumDisks >= 1);
DiskCm: AnyOf MyCluster where HasDisk = True;
```

Next we determine the number of processors available.

```
CmCount : NumberOf MyCluster;
```

The module's construction specification includes code to create the server processes. The **CmCount** value is used to control the number of server processes instantiated. This is the only use of hardware attributes that is legal within template construction sections.

```
(i=1..CmCount max 6) ServerPM[i] : Process . . .
```

The subsequent resource directives in the directives section of the template are

```
Same (DiskCm, DiskManagerPM);
NearCm ((i=1..6) ServerPM[i]);
NearCm (DiskManagerPM, (i=1..6) ServerPM[i]);
```

These directives specify assignment of processes to processors. Nothing is stated about where the module and process components should be placed into memory. If desired, such directives can be added; otherwise, the components are placed into memory randomly.

The example can be summarized as—

```
Module BossMod is
Construct ( (i=1..CmCount max 6) ServerPM[i] : Process . . .
                DiskManagerPM : Process . . .
                BossPM : Process . . .
                . . . )
. . .
Directives (
        MyCluster: AnyOf CmStar where (NumCms >= 2 And NumDisks >= 1);
        DiskCm: AnyOf MyCluster where HasDisk = True;
        CmCount: NumberOf MyCluster;

        Same (DiskCm, DiskManagerPM);
        NearCm ((i=1..6) ServerPM[i]);
        NearCm (DiskManagerPM, (i=1..6) ServerPM[i]);
    )
```

Example 6-5: TASK: Use of Directives

Note that so far only single hardware components have appeared within directives. There is exactly one case in which we allow hardware sets to be named directly within a resource directive. Names of resource sets appear in directives only when an iteration construct precedes the directive. Thus, instead of the directive

```
NearCm ((i=1..6) ServerPM[i]);
```

we could have written

```
(i=1..6) Same (MyCluster[i], Server[i]);
```

No variable upper bounds are allowed when iterated directives are used. Instead, the maximum values must be specified within the construction parts of templates. Consider the creation of a variable number of processes.

```
(i=1..n max 6) ServerPM[i] : Process . . .
```

Then the following statement is required in the directives section:

```
NearCm ((i=1..6) ServerPM[i]);
```

When such maximum values are used, it is understood by the compiler that the actual rather than the maximal number of processes should be employed for resource allocation. Note that the compiler does not consider `CmCount` a variable upper bound, since its value can be determined at compile-time.

A.11. Invoking the TASK Compiler

The TASK compiler is invoked by the command

```
.ru task[x335cm01]
```

on CMU-10A. This section lists and describes the commands which may be issued to the TASK compiler. It is an edited version of the file `task.hlp[x335cm01]`, and can be viewed by typing "`help`" to the compiler.

Architecture *<Architecture Description File>*

The **Architecture** command sets the architecture description file. The standard description is in **ARCHIT.STD[x335cm01]**. User-written descriptions files must have the same format.

Compile *<Task Program File>*

The **Compile** command runs the TASK compiler for the named program. After compilation, CCL linkage is used to assemble the .tem files produced by TASK (loader command files). The .tem files will become .obj files that are included in the linker command files produced by TASK. For each task force, the following files are generated:

Error File:

<TaskProgramName>.ERR

Definitions Files (to be compiled along with the BLISS code):

<TaskForceName>.DFS
<ModuleName>.DFS—one per module
*<ModuleName>*_*<FctIndex>*.DFS—one per function per module.

Linker Command Files:

<TaskForceName>.LMD—the linker command file for the entire task force.
<TaskForceName>.MIC—a command file to run the linker.
<ModuleName>.LMD—one per module

Loader Command Files:

<TaskForceName>.TEM (the non-assembled version)
<TaskForceName>.OBJ (the assembled version)
<ModuleName>.TEM/.OBJ—one per module

Directory *(Not yet implemented)*

The **Directory** command shows a directory of files produced during the current run of the TASK skeleton. Each **Compile** command appends a new directory record. Currently, for each task force only the loader command files are shown.

DiscardFiles

The **DiscardFiles** command prompts TASK to delete all intermediate files produced during a TASK run.

Exit

The **Exit** command is used for leaving the program.

Gripe *<Subject>*

The **Gripe** command provides direct contact between the users and maintainer of the program. The gripe message is mailed to the current maintainer of the program who should respond within a reasonable period of time.

Help *<Command Name>*

The **Help** command types a one page description of the given command. If *<Command Name>* is empty, this text is given. For a list of commands use the **?** command. For examples of running TASK programs, look at one of the following files:

```
Skelet.Tsk[x336cm01]
PDE.Tsk[x336cm01]
Test.Tsk[x336cm01]
NTest.Tsk[x336cm01] (has proximity directives)
LTest.Tsk[x336cm01]
Monito.Tsk[x336cm01]
```

Note: Input may come from command files. A command file is introduced by "**@<File name>**" instead of a command. Command files may be nested to a depth of 10. Lines beginning with ";" are treated as comments.

KeepFiles

The **KeepFiles** command prompts TASK to refrain from deleting the intermediate files produced during a TASK run.

Mode

The **Mode** command displays the current switch values.

News [*]

The **News** command will give a description of recent changes to TASK. "**News ***" will give all the changes listed in the log file.

NoOptimization the default

The **NoOptimization** command turns of all optimization of resource usage for a given TASK program. The loader will place objects and assign processes wherever space or processors are available.

NoResults the default

The **NoResults** command optimizes the task force without displaying the results of optimization.

OnceCompile *<Task Program Name>*

The **OnceCompile** command compiles a TASK program and exits after compilation is complete. The **Compile** command, by contrast, returns to TASK after compilation of a program to allow multiple program compilations in one TASK run.

OptAlternate turned off by default

The **OptAlternate** command turns on the "alternate" optimization policy provided by TASK. This policy uses the same information for optimization that is employed by regular optimization. However, in this case, process assignment is done before data assignment to processes.

Optimization turned off by default

The **Optimization** command instructs TASK to optimize the use of resources of the given TASK program. The loader is instructed to use specific computer modules for each object being placed and each process being assigned. The user will be asked whether he wishes resource directives to be treated as constraints or as preferences. Temporarily, "0" or "*<Return>*" mean "Preferences" and 1"

means "Constraints". The latter make sure that such directives as "DiffCM" really assign or place things in different Cms; the former just result in the compiler "giving it a try" but doing something else if deemed appropriate.

Proximities the default for optimization

The **Proximities** command instructs TASK optimization to optimize resource usage based on the component proximities specified with TASK resource directives. Default proximities are used if no directives are specified.

Quit

Same as **Exit**.

Results

The **Results** command induces the compiler to display tailoring results. The user is prompted for whether results should be displayed at the terminal or written into a file.

RunTimes

The **RunTimes** command instructs TASK to optimize resource usage based on process run times for process assignment and proximities for data placement. By default, process run times are not used.

Selections

The **Selections** command instructs TASK to "optimize" by simply satisfying the selections specified in TASK resource directives. If no selections are specified, arbitrary resources are used by the loader.

`SetErrorOption` *<Option>*

By using the `SetErrorOption` command before a `Compile` command the user may direct compilation errors to the terminal, to a file, or both.

?

The ? command produces a list of the available commands.

;

The ; command is a comment facility. Anything can be written, on a TASK command line following ;.

A.12. A Full TASK Program

The following example is the complete task-force description for the Skeleton program.

```
!-----------------------------------------------------------------
!
! Task force: SkForce
!
!       The skeleton process task force consists consists of one
!       module with one function:
!
! Module: SkMod(ule)
!
! Function 0: SkeletonProcess
!
! This file: Skelet.Tsk[x335cm01] CopyRight (c) 1981, Karsten Schwans
!
!-----------------------------------------------------------------

TaskForce SkForce Is
Construct (SkModule: New SkMod;)
Directives ()

Module SkMod Is
Construct (
        ! The debugger expects that code objects
        ! are found in slots in the module that correspond to the
        ! window slots into which code is loaded. We therefore put
        ! something random into the module slot 0.
        Slot0:  NAME;
        ! The first object contains owns used by the skeleton
        ModCode1: New Basic (Source = ("Stack[x3351d20]",
                                       ! If using debugger:
                                       "612u[x335cm0s]<(g) u612>",
                                       "Skeleton[x335cm01](GO)",
                                       ! Your program's owns here:
                                       ! "YourProgram(GO)",
                                       "UserIO[x335cm0r](GO)",
                                       "SavReg[1130b198](GO)",
                                       "SigEnb[x3351d20](GO)"));
        ! The skeleton process code
        ModCode2: New Basic (Source = ( ! Task version of skeleton
                                       "Skeleton[x335cm01](CP)",
                                       ! If using the debugger :
                                       "Initial[x335cm0s]",
                                       "612u[x335cm0s]<(CP) u612>",
                                       "612k[x335cm0s]",
                                  "linktb[x335cm0s]<(C) link>"));
        ! Code that everyone needs to use
        ModCode3: New Basic (Source = ( "savreg[1130b198](CP)",
                                       "sigenb[x3351d20](CP)",
                                       "userio[x335cm0r](CP)"));

        ! The invocation mailbox for 'present' skeleton function
        InvokeMB: New Mailbox (MsgType = "Capa");

        ! The next files needed only if the debugger is used
        ! The 'name' csects
        Names: New Basic (Source [0, "/N"] =
```

```
                                        ("Skeleton[x335cm01](N)",
                                        ! "YourProgram(N)",
                                        "Star[x335cm0s](N)",
                                        "predef[1130b198]<(N) K$BL>",
                                        "612u[x335cm0s]<(N) u612>"));
        ! The 'symbol' csects
        Symbols: New Basic (Source [1, "/S"] =
                                        ("Skeleton[x335cm01](S)",
                                        ! "YourProgram(S)",
                                        "Star[x335cm0s](S)",
                                        "predef[1130b198]<(S) K$BL>",
                                        "612u[x335cm0s]<(S) u612>"));
! The skeleton process
        SkProcess: Process SkMod . Skeleton ();
        )

Function Skeleton (Present [InvokeMB]) Is
Construct (
        ProcessStack:   New Basic (Stack, StackOwns [ModCode1]);
        ! The debugger forces us to explicitly allocate windows
        ! for the code. InitialCode is always in window1.
        Code2:          Ref ModCode2 (InitialCode["SkeletonProcess"]);
        Code3:          Ref ModCode3 (Window[2]);
        InvokeWindow:   Name(Window);
        ProcessCarrier: Name(Window);
        MessageRock:    Name;
        ! YourName:     Name;
        ! YourObject:   New Basic;
        ! . . .;
        )
Directives ()

Directives ()
```

Example 6-6: TASK Description for the Skeleton

Appendix B
TASK Grammar

Expressions, Names, and Types

⟨Simple Name⟩ ::= *⟨Unquoted String⟩*

⟨Simple Template Name⟩, *⟨Complex Template Name⟩* ::= *⟨Simple Name⟩*
Template names must be unique in their first five characters.

⟨Formal Parameter Name⟩, *⟨Var Name⟩* ::= *⟨Simple Name⟩*

⟨Comp Name⟩ ::= *⟨Simple Name⟩*

⟨Function Name⟩ ::= *⟨Complex Template Name⟩* . *⟨Complex Template Name⟩*
⟨Object Name⟩ ::= *⟨Simple Name⟩* | *⟨Access Expr⟩*

⟨Path Name⟩ ::= *⟨Simple Name⟩* . *⟨Path Name⟩* | *⟨Simple Name⟩*
⟨Object Path Name⟩ ::= *⟨Object Name⟩* . *⟨Object Path Name⟩* | *⟨Object Name⟩*

⟨Keyword Name⟩ ::=
⟨Obj Common Att⟩ | *⟨Obj Add Att⟩* | *⟨Obj Special Att⟩* | *⟨Hard Att⟩*

⟨Var Type⟩ ::= **String** | **Integer** | **Boolean**

⟨Expr⟩ ::= *⟨Arith Expr⟩* | *⟨Quoted String⟩* | **True** | **False**

Templates

⟨*Templates*⟩ ::= ⟨*Template*⟩* EOF

⟨*Template*⟩ ::= {⟨*Complex Template*⟩|⟨*Simple Template*⟩}

⟨*Simple Template*⟩ ::=
⟨*Simple Object Type*⟩⟨*Simple Template Name*⟩ (⟨*Actual Attributes*⟩)#

Some actual attributes may occur only within functions or modules. Hence there are semantic restrictions concerning which attributes may appear in simple templates that are not bound to a particular complex template.

⟨*Complex Template*⟩ ::= ⟨*Task-Force Description*⟩
|⟨*Module Description*⟩
|⟨*Complex-Basic Description*⟩

⟨*Simple Object Type*⟩ ::= **Basic** | **Stack** | **Mailbox** | **Deque** | **Device**

Complex Templates

<Module Description> ::=
Module *<Complex Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
<Function Description>+
<Resource-Usage Directives>

All module attributes must be specified at declaration time. Modules can only be components of task-force templates. Only one instance of a module can be constructed from a particular template.

<Function Description>::=
Function *<Complex Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
<Resource-Usage Directives>

A *<Function Description>* can only be a component of a *<Module Description>*.

<Task-Force Description> ::= **TaskForce** *<Complex Template Name>* **is**
<Construction Description>
<Resource-Usage Directives>

A task force template can have no parameters. A *<Task-Force Description>* cannot be a component of another template. No more than one task force template may appear in one TASK program.

<Complex-Basic Description> ::=
ComplexBasic *<Complex Template Name>* (*<Formal Parameters>*)# **is**
<Construction Description>
<Resource-Usage Directives>

Components may have any type except those specifically excluded above.

Construction Descriptions

⟨Construction Description⟩ ::= **Construct** (⟨Component⟩ ;*)

⟨Component⟩ ::= ⟨Comp Name⟩ : ⟨Operation⟩
| ⟨Iteration⟩ : ⟨Operation⟩
| ⟨Expanded Iteration⟩ : ⟨Operation⟩ [not yet implemented]

⟨Operation⟩ ::= **New** {{⟨Object Type⟩ (⟨Actual Parameters⟩)#}
| ⟨Template Name⟩ (⟨Actual Parameters⟩)#)}
| **Reserve** {{⟨Object Type⟩ (⟨Actual Parameters⟩)#}
| ⟨Simple Template Name⟩ (⟨Actual Parameters⟩)#}
| **Ref** ⟨Object Name⟩ (⟨Actual Attributes⟩)#
Only special attributes should be used here.
| **Use** ⟨Object Name⟩ (⟨Actual Attributes⟩)#
| **Process** ⟨Function Name⟩ (⟨Actual Parameters⟩)
Note: returns a mailbox. Semantics: Processes may be created only within the body of the module defining their function
| **Name** (⟨Actual Attributes⟩)#
Only the **Window** attribute should appear here.

Iterations

⟨Iteration⟩ ::= (⟨IterName⟩ = ⟨Low Limit⟩ . . ⟨High Limit⟩) ⟨Access Expr⟩

⟨Access Expr⟩ ::= ⟨Iterated Name⟩ [⟨Index⟩]

⟨Index⟩ ::= ⟨IterName⟩ {**Mod** ⟨Var Expr⟩}#
⟨Low Limit⟩ ::= ⟨Integer⟩
⟨High Limit⟩ ::= ⟨Integer⟩
| * **Max** ⟨Integer⟩
| ⟨Integer⟩ **Max** ⟨Integer⟩
| ⟨Variable⟩

⟨IterName⟩ ::= ⟨Simple Name⟩
⟨Iterated Name⟩ ::= ⟨Comp Name⟩
| ⟨Formal Parameter Name⟩
| ⟨Actual Parameter Name⟩

⟨Expanded Iteration⟩ ::= ⟨Comp Name⟩,+
Expanded iteration and arithmetic expressions are not implemented.

Parameters and Attributes

⟨Formal Parameters⟩ ::= ⟨Formal Parameter⟩* | ⟨Actual Attribute⟩*
⟨Formal Parameter⟩ ::=
{⟨Formal Parameter Name⟩,+ | ⟨Iteration⟩} : ⟨Simple Template⟩,
| ⟨Var Name⟩,+ : ⟨Var Type⟩,

⟨Actual Attributes⟩ ::= ⟨Actual Attribute⟩*
⟨Actual Attribute⟩ ::= {⟨Keyword Name⟩ = {⟨Expr⟩ | ⟨Var Expr⟩ | ⟨Index⟩},}
| **Source** {[⟨Integer⟩, ⟨Link Switches⟩]}#
= ⟨Source Parameter Value⟩,
| **Rights** = (⟨Quoted String⟩, ⟨Quoted String⟩),
Currently, all rights specifications remain unused.
| **Size** = (⟨Integer⟩, ⟨Integer⟩),
| ⟨Special Attr⟩,

⟨Actual Parameters⟩ ::= {⟨Actual Parameter⟩* | ⟨Actual Attribute⟩*}
⟨Actual Parameter⟩ ::= ⟨Key Expr⟩ = ⟨Actual Expr⟩,

⟨Actual Expr⟩ ::= ⟨Object Name⟩ | ⟨Iteration⟩ | ⟨Expr⟩
| ⟨Var Expr⟩ | ⟨Access Expr⟩ | ⟨Index⟩

⟨Key Expr⟩ ::= ⟨Formal Parameter Name⟩ | ⟨Access Expr⟩ | ⟨Iteration⟩

⟨Source Parameter Value⟩ ::= ({⟨Quoted String⟩},+)
⟨Link Switches⟩ ::= ⟨Quoted String⟩
One or more switches (/D, /N, etc.) to be passed to the Cmmp linker.

⟨Obj Common Att⟩ ::= **Source** | **Rights** | **Size**
⟨Obj Add Att⟩ ::= **StackSize** | **Class** | **Preempt** | **Quantum**
| **ServiceLimit** | **ProcessId** | **MsgType**
For a description of these attributes, see Appendix C.

⟨Obj Special Att⟩ ::= **Window** | **Stack** | **InitialCode**
| **PrivateMailbox** | **StackOwns** | **Alias** | **Present**

⟨Special Attr⟩ ::= **Window** [⟨Integer⟩]#
| **PrivateMailbox**
| **InitialCode** [⟨Quoted String⟩]#
| **Stack**
| **StackOwns** [⟨Object Name⟩]
| **Present** [⟨Object Name⟩]
| **Alias** [" ⟨Function Name⟩ "]

Resource-Usage Directives

⟨Resource-Usage Directives⟩ ::= **Directives** (⟨Selection⟩;* ⟨Directive⟩;*)

⟨Selection⟩ ::= {⟨Hardware Set Name⟩ | ⟨Var Name⟩} : ⟨Selection Expr⟩
⟨Directive⟩ ::= ⟨Iteration⟩# ⟨Proximity Degree⟩
({⟨Iteration⟩ # ⟨Object Path Name⟩)},+)
| ⟨Iteration⟩ # **Same** (⟨Hardware Set Name⟩ [⟨Index⟩]#,
{⟨Iteration⟩ # ⟨Object Path Name⟩},+)
In the first implementation no paths will be used.

⟨Proximity Degree⟩ ::= ⟨Integer⟩ | **SameCm** | **SameCluster** | **NearCm**
| **DifferentCm** | **DifferentCluster** | **NoCare**
⟨Selection Expr⟩ ::= ⟨Set Expr⟩ {**where** ⟨Attr Expr⟩}#
| **AnyOf** ⟨Set Expr⟩
| **NumberOf** ⟨Set Expr⟩

⟨Set Expr⟩ ::= ⟨Set Name⟩ | '(⟨Set Name⟩ ⟨Opr⟩ ⟨Set Name⟩)
⟨Set Name⟩ ::= ⟨Predefined Set Name⟩ | ⟨Hardware Set Name⟩
⟨Attr Expr⟩ ::= ⟨Hard Att⟩ ⟨RelOpr⟩
| (⟨Hard Att⟩⟨RelOpr⟩⟨Hard Att Value⟩⟨Opr⟩⟨Hard Att⟩
⟨RelOpr⟩⟨Hard Att Value⟩)#

⟨Opr⟩ ::= **and** | **or**
⟨RelOpr⟩ ::= < | > | = | <= | >=

⟨Hard Att Value⟩ ::= ⟨Integer⟩ | **True** | **False**
⟨Hard Att⟩ ::= **MaxSize** | **ActualSize** | **Reliability**
| **Speed** | **NumEther** | **NumLines** | **NumDisks**
| **NumDALinks** | **NumCms** | **NumClusters**
| **HasDALink** | **HasDisk** | **HasEther** | **HasLine**
| **MaxMPSize** | **ActualMPSize** **MPReliability**
| **PCReliability** | **PCSpeed**

⟨Predefined Set Name⟩ ::= **Cm** | **CmStar**
| **Cluster0** | **Cluster1** | ... | **Cluster4**
| **Cm[0,0]** | **Cm[0,1]** | ... | **Cm[0,14]**
| **Cm[1,0]** | **Cm[1,1]** | ... | **Cm[1,14]**
| **Cm[2,0]** | **Cm[2,1]** | ... | **Cm[2,14]**
| **Cm[3,0]** | **Cm[3,1]** | ... | **Cm[3,14]**
| **Cm[4,0]** | **Cm[4,1]** | ... | **Cm[4,14]**

⟨Hardware Set Name⟩ ::= ⟨Simple Name⟩

Appendix C
Parameters for TASK Templates

Templates in the TASK language may have a variety of parameters. Different parameters are defined for different object types. This appendix lists each object type, and then the parameters that are defined for it. The Size parameter for some object types takes an ordered pair, enclosed in parentheses. For some parameters, default values are used if the parameter does not appear. Defaults are given in italics after the description of the parameter.

Basic Objects

Size (Number of bytes in data part, Number of slots in capability part) *(4096, 0)*

Source The file(s) from which the data part is to be initialized. *(No initialization performed)*

Window[*<window number>*] If this parameter is present, the object will be loaded into window *<window number>* when the process is created. This parameter may be specified only within a function body.

InitialCode, Stack, StackOwns Defined below, on page 204.

Mailboxes

Size (Maximum number of registered receivers, number of messages that may be stored in mailbox) *(32, 128)*

MsgType Type of mailbox, "*Capability*" or "*Data*". *(Data)*

PrivateMailbox Defined below

Stack and Deque Objects

Size (Number of slots in capability part, maximum number of (data) entries which may be stored in stack or deque) *(128, 128)*

Functions

Value Parameters, specified as attribute parameters:

Process ID It is possible to specify a value for the process ID, rather than letting the ***Process Creator*** use its running counter. If this parameter is specified, its value is used as the Process ID. If it is not specified, the ***Process Creator*** uses its counter. (This is the only process parameter that can be a TASK variable; and, if not specified, it is the only process parameter for which a default value will be used).

Size Number of capabilities in the *process name space*. *(32)*

StackSize Size of the process stack, in bytes. *(4096)*

Present[*<Object Name>*] Indicates that this is a present function. The object name must refer to a mailbox in the body of the module template which can be used as the process's invocation mailbox.

Quantum Size of the time quantum to be used, if this process executes in a time-sliced environment. *(0 line-frequency clock ticks)*

ServiceLimit Maximum execution time the process will receive. *(0 line-time clock ticks)*

Class The class value used by the scheduler.

Preempt A boolean indicating whether or not this process should cause preemption of the preferred processor when it is sent to its run queue. *(False)*

Object Attributes, specified within the body of the function, in conjunction with selected component objects:

InitialCode[*<RoutineName>*] The code object that is to be loaded and executed when the process is first run, followed by the name of the BLISS routine that contains the entry point, quoted, within brackets.

Stack

The specification of the process's stack object. If this is omitted, a default stack with the size indicated above (see StackSize parameter) will be created.

StackOwns [*<Object Name>*] The specification of the object that contains the own values that are to be put onto the stack.

PrivateMailbox A mailbox private to a new process. If omitted, a default private mailbox is provided by the ***Process Creator***.

When a function is defined, its attributes can be set so that every process instantiated for that function inherits them. Alternatively or in addition, the process creation statement can list attributes.

Module

Size Number of capabilities in the module name space. *(32)*

Task Force

Size Number of capabilities in the task-force object. *(32)*

Appendix D
Process Creator and Loader

D.1. Process Creator

The *Process Creator* module provides functions to create new processes and, despite its name, to terminate existing processes. New processes are created under two sets of circumstances. In the first case, if an ***Invoke*** instruction requires the creation of a new process, then the *Nucleus* of the STAROS system will send the carrier to the invocation mailbox [149] for function 0 of the process creator module. A new process will be created, and the carrier forwarded to either the invocation mailbox of a *present* function or, if the function is *absent*, to the private mailbox of the new process. In the second case, the user may invoke function 1 of the module as an explicit request to create a new process. In this case, a new process will be created, and capabilities for the private mailbox and the process object will be returned to the user in the carrier. The user can transmit a carrier to the process by sending a capability for the carrier to the private mailbox of the new process.

Function 2 of the process creator module terminates an existing process by setting the proper state within the process object and removing the process from the process set of its module. The nucleus invokes this function as the result of a ***Terminate*** instruction or in the case of a severe error condition that precludes allowing the process to continue to execute.

D.2. Loader

The single function of the *Loader* module creates a general tree-structured collection of objects. Specification of the objects is contained in a file which includes *command templates* describing the objects to be created and *command lists* with instructions for the initialization of the new objects. In general, a command template may point to a command list, and a command list may include pointers to command templates for additional objects to be constructed. Commands may also specify that a

capability for a new object is to be placed in the *loader library*. Capabilities from the loader library may be specified for the initialization of objects.

The parameters for invoking the loader are a pair of ASCIZ strings: a *host name* and a *file name*. The host name identifies some file system, and the file name is an appropriate identifier for some file in the file system. If the file is successfully loaded, a capability for the root of the tree of objects is returned.

6.0.1. Command Templates

A command template specifies the creation of an object. In the file to be loaded, each field, except the last, in every template record is a *loader parameter value*, described below. The last field is a pointer within the file to an optional command list for the initialization of the object.

6.0.1.1. Command Templates for objects other than processes:

Data Size	The number of bytes in the data part of the new object.
Capa Size	The number of capas in the capa part of the new object.
Entry Size	The number of entries in the new object.
Locality	Where the new object is to be created (what computer module).
Type	The type of the new object.
DataP	Whether is data part of the object is to be initialized from a file produced by the link editor.
Command List	A pointer within the file to an optional command list.

6.0.1.2. Command Template parameters for processes:

Function Number	The function number within module of the process.
Locality	The run queue to which the process will be assigned.
Module reference	The module for the process.
ProcessId	Value for the *ProcessId* field of the process.

Command List	A pointer to an optional command list.

6.0.2. Command Lists

A command list is the concatenation of single-word commands, each of which may be followed by a list of parameters.

Call	This object is to receive a capability for yet another object. Parameters:
Template	A pointer to a template (not a process template).
Slot	The slot within this object where the capability is to be placed, a loader parameter value.
Parameter list	A list of loader parameter values to be substituted for *parameter* type loader parameter values within the template.
CopyCapability	Copy a capability. Parameters:
Source	Either an integer index into the object being initialized, or else a loader parameter value of either type *capability index* or type *stack capability*.
Destination	Either an integer index into the object being initialized, or else a loader parameter value of either type *capability index* or type *stack capability*.
CopyObject	Make a copy of an indicated object. Parameters:
Source	Either an integer index into the object being initialized, or else a loader parameter value of either type *capability index* or type *stack capability*.
Destination	Either an integer index into the object being initialized, or else a loader parameter value of either type *capability index* or type *stack capability*.
Locality	Where the copy is to be created
CreateProcess	Invoke the ***Process Creator*** module to request the creation of a new process.
Template	A pointer to a process template.
Parameter list	A list of loader parameter values to be substituted

for *parameter* type loader parameter values within the template.

Done — The end of a command list. No parameters.

EnterLibrary — A capability for the object is to be stored in the loader library. Parameter:

- Slot — The slot within the loader library.

For — Initialize a list of slots within the object. Parameters:

- Template — A pointer to a template.
- Slot — The first slot within this object where a capability is to be placed, a loader parameter value.
- HowMany — The number of slots to be initialize, a loader parameter value.
- Parameter list — A list of loader parameter values to be substituted for *parameter* type loader parameter values within the template.

6.0.3. Loader Parameter Values

Loader parameter values are 32-bit quantities where the high-order byte of the first word indicates what type of thing is being specified. The types of loader parameter values are:

Capability index — A two-part index: an index into the stack of objects being initialized, and an index into the capability part of the selected object.

End Marker — The end of a list of parameters.

Integers — A 32-bit integers using the type codes **#000** and **#377**.

Library Index — An index into the loader library.

Parameter — When a template is interpreted, a value from *parameter list* of the command will be substituted for each parameter type loader parameter value. The low-order byte of the first word indicates which item from the parameter list is to be substituted. The 16-bit signed interger taken from the second word is added to the value to be substituted.

Stack Capability — an index into the stack of objects being initialized.

Cm type — The specification of a particular type of Cm. A Cm and its cluster are characterized by four pairs of bits. For each of four devices (Ethernet, disk, tty and DAlink) two bits indicate whether the Cm and cluster to be selected *must* have such a device, *must not* have such a device, or *need not* have such a device.

Bibliography

[1] S.R. Arora and A. Gallo.
Optimization of Static Loading and Sizing of Multilevel Memory Systems.
Journal of the Assoc. of Comp. Mach. 20(2):307-319, April, 1973.

[2] M.M. Astrahan, D.D. Chamberlin.
Implementation of a structured English Query Language.
Comm. of the Assoc. Comput. Mach. 18(10), Oct., 1975.

[3] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes.
The Illiac IV computer.
IEEE Trans. Comput. C-17(8):746-757, Aug., 1968.

[4] A.L. Barrese and S.D. Shapiro.
Structuring Programs for Efficient Operation in Virtual Memory Systems.
IEEE Trans. Soft. Eng. SE-5(6):643-652, Nov., 1979.

[5] Forest Baskett, K.M. Chandi, Richard R. Muntz, and Fernando G. Palacios.
Open, Closed, and Mixed Networks of Queues with Different Classes of Customers.
Journal of the Assoc. Comput. Mach. 22(2):248-260, April, 1975.

[6] G. M. Baudet.
The design and analysis of algorithms for asynchronous multiprocessors.
PhD thesis, Carnegie-Mellon University, 1978.

[7] Gerard M. Baudet.
The Design and Analysis of Algorithms for Asynchronous Multiprocessors.
PhD thesis, Computer Science Department, Carnegie-Mellon University, April, 1978.

[8] D.M. Berry and M.H. Penedo.
The Use of Module Interconnection Specification Capability in the SARA Design methodology.
In *Proceedings of the Fourth International Conference on Software Engineering*, pages 294-307. IEEE,ACM,GI, Sept., 1979.

[9] Wm.A. Wulf et al.
Bliss-11 Manual.
Technical Report, Computer Science Dept., Carnegie-Mellon Univ., 1975.

[10] S.H. Bokhari.
Dual Processor Scheduling with Dynamic Reassignment.
IEEE Trans. Soft. Eng. SE-5(4):341-349, July, 1979.

[11] T.L. Booth and C.A. Wiecek.
Performance Abstract Data Types as a Tool in Software Performance Analysis.
IEEE Trans. Soft. Eng. SE-6(2):138-151, March, 1980.

[12] W.C. Brantley, G.W. Leive, and D.P. Siewiorek.
Decomposition of data flow graphs on multiprocessors.
In *Proceedings of the National Computer Conference, NCC*, pages 281-290. Assoc. Comput. Mach., 1977.

[13] Per Brinch Hansen.
The Programming Language Concurrent Pascal.
IEEE Trans. Soft. Eng. SE(1):199-207, 1975.

[14] Per Brinch-Hansen.
Distributed Processes: A Concurrent Programming Concept.
Comm. of the Assoc. Comput. Mach. 21(11), Nov., 1978.

[15] Per Brinch-Hansen.
Joyce, A Language for Computer Networks.
Technical Report, Computer Science Department, University of Southern California, Nov., 1979.

[16] R.S. Brooks and V.R. Lesser.
Network Automotive Traffic Light Control.
Technical Report, COINS Technical Report 79-13, University of Amherst, Amherst, Mass., Feb., 1979.

[17] Mark E. Brown and Bruce W. Weide.
Preliminary Design of a Highly Parallel Architecture for Real-Time Applications.
In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computation, Univ. of Illinois.* Department of Computer and Information Science, The Ohio State University, Oct., 1980.

[18] R.G. Casey.
Allocation of Copies of a File in an Information Network.
In *NCC Conference Proceedings.* Assoc. Comput. Mach., 1972.

[19] R.G. Casey.
Design of tree networks for distributed data.
In *National Computer Conference*, pages 251-257. Assoc. Comput. Mach., 1973.

[20] Xavier Castillo.
A Compatible Hardware/Software Reliability Prediction Model.
Technical Report, Department of Electrical Engineering, Carnegie-Mellon University, July, 1981.

[21] R.J. Chansler and Pradeep Sindhu.
The Image Application on Cm*.
1977.
Internal Memo of the StarOS Group, Computer Science Dept., Carnegie-Mellon Univ.

[22] Peter S. Chen.
Optimal file allocation in multi-level storage systems.
In *Proceedings of the National Computer Conference, NCC*, pages 277-281. Assoc. Comput. Mach., 1973.

[23] N.F. Chen and C.L. Liu.
On a Class of Scheduling Algorithms for Multiprocessor Computing Systems.
In *Proceedings of the Conference on Parallel Processing*, pages 1-16. Assoc. Comput. Mach., IEEE, 1975.

[24] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager.
Thoth, a Portable Real-Time Operating System.
Comm. of the Assoc. Comput. Mach. :105-115, Feb., 1979.

[25] Wesley W. Chu.
Optimal File Allocation in a Multiple Computer System.
IEEE Trans. on Computers C-18(10):885-889, Oct., 1969.

[26] Wesley W. Chu.
Performance of file directory systems for data bases in star and distributed networks.
In *Proceedings of the National Computer Conference*, pages 577-587. Assoc. Comput. Mach., 1976.

[27] Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lan, and Kemal Efe.
Task Allocation in Distributed Data Processing.
Computer Magazine 13(11):57-70, Nov., 1980.

[28] Eds. I. Durham, S.F. Fuller, and A.K. Jones.
The Cm Review Report.*
Technical Report, Comp. Science Dept., Carnegie-Mellon Univ., 1977.

[29] Eds. A.K. Jones and Ed Gehringer.
The Cm Multiprocessor Project: A Research Review.*
Technical Report, Comp. Science Dept., Carnegie-Mellon Univ., CMU-CS-80-131, July, 1980.

[30] Roy Levin et al.
The C.mmp Linker Reference Manual.
Technical Report, Computer Science Dept., Carnegie-Mellon Univ., 1975.

[31] CMU.
Speech Understanding Systems: Summary of Results of the Five-Year Research Effort at Carnegie-Mellon University.
Technical Report, Department of Computer Science, Carnegie-Mellon University, Aug., 1977.

[32] E.G. Coffmann, Jr., E. Gelenbe, and B. Plateau.
Optimization of the Number of Copies in a Distributed Data Base.
IEEE Trans. Soft. Eng. SE-7(1):78-84, Jan., 1981.

[33] Cook, R.P.
*Mod - A Language for Distributed Programming.
In *Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama*, pages 223-241. IEEE, Oct., 1979.

[34] William Corwin.
The Scheduling of Primary Memory in a Multiprocessor.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1982.
In preparation.

[35] Norbert Cullmann.
Load-Sensitive Software Distribution in Satellite Graphics Systems.
In *Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama*, pages 72-78. IEEE, Oct., 1979.

[36] O.-J. Dahl, E.W.Dijstrat, C.A.R. Hoare.
Structured Programming.
Academic Press, 1972.

[37] Peter J. Denning.
The Working Set Model for Program Behaviour.
Comm. of the Assoc. Comput. Mach. 11(5), May, 1968.

[38] P.J. Denning.
Working Sets: Past and Present.
IEEE Trans. on Software Engineering SE-6(1):64-84, Jan., 1980.

[39] J.B. Dennis, J.B. Fosseen, J.P. Linderman.
Data Flow Schemas.
Technical Report, Project MAC, M.I.T., Cambridge, MA, July 27, 1972.

[40] Frank DeRemer and Hans H.Kron.
Programming-in-the-Large vs. Programming-in-the-Small.
IEEE Trans. on Software Eng. SE-2(2):80-86, June, 1976.

[41] O. El-Dessouki, W. Huen, M. Evens.
Towards a Partitioning Compiler for a Distributed Computing System.
In *Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama*, pages 296-304. IEEE, Oct., 1979.

[42] I.Durham, R.C. Dugan, A.K. Jones., and S.N. Talukdar.
Power System Simulation on a Multiprocessor.
In *Text of Abstracts of the IEEE Power Engineering Society Summer Meeting, Vancouver, British Columbia, Canada*. July 15-20, 1979.

[43] England, D. M.
Capability concept mechanisms and structure in System 250.
In *International Workshop on Protection in Operating Systems*, pages 63-82. IRIA/LABORIA, Rocquencourt, France, August 13-14, 1974.

[44] Ph.H. Enslow, Jr.
What is a Distributed Processing System ?
IEEE Computer 11,1, Jan., 1978.

[45] G. Estrin.
SARA Aided Design of Software for Concurrent Systems.
In *AFIPS Proceedings, National Computer Conference, Anaheim, CA, Volume 47*, pages 337-347. June, 1978.

[46] G. Estrin.
A Methodology for the Design of Digital Systems: Supported by SARA at the Age of One.
In *AFIPS Proceedings, National Computer Conference, Anaheim, CA, Volume 47*, pages 313-324. June, 1978.

[47] K. P. Eswaran.
Placement of Records in a file and File Allocation in a Computer Network.
Proceedings of the IFIP 74 , 1974.

[48] Peter H. Feiler.
An Interactive Language-Oriented Programming Environment Based On Compilation Technology.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, May, 1982.

[49] P. Feiler, Wm. Wulf, Zinnikas, R. Brender.
A Quantitative Technique for Comparing the Quality of Language Implementations.
1981.
In preparation.

[50] J.A. Feldman.
High Level Programming for Distributed Computing.
Comm. of the Assoc. Comput. Mach. 22(6), June, 1979.

[51] R.D. Fennel and V.R. Lesser.
Parallelism in AI Problem Solving : A Case Study of HearSay II.
Technical Report, Comp. Science Dept., Carnegie-Mellon Univ., Oct., 1975.

[52] Domenico Ferrari.
Improving Locality by Critical Working Sets.
Comm. Assoc. Comput. Mach. 17(11):614-620, Nov., 1974.

[53] Micheal J. Fischer, Nancy D. Griffeth, Leo J. Guibas, Nancy A. Lynch.
Optimal Placement of Identical Resources in a Distributed Network.
Technical Report, Georgia Institute of Technology, School of Information and Computer Science, Oct., 1980.

[54] M.J. Flynn.
Very High-Speed Computing Systems.
Proceedings of the IEEE 54(12), 1966.

[55] Derrell v. Foster, Lawrence W. Dowdy, and James E. Ames, IV.
File Assignment in a Star Network.
In *Proceedings of the SIGMETRICS/CMG VIII Performance Conference, Washington, D.C.*, Assoc. Comput. Mach., Dec., 1977.

[56] H. Frank and W. Chou.
Topological Optimization of Computer Networks.
IEEE Trans. on Communication , June, 1972.

[57] S.H. Fuller, J.K. Ousterhout, L. Raskin, P.I. Rubinfeld, P.J. Sindhu, and R.J. Swan.
Multiprocessors : An Overview and Working Example.
Proc. of the IEEE 66(2), Feb., 1978.

[58] M.R. Garey and D.S. Johnson.
Complexity Results for Multiprocessor Scheduling under Resource Constraints.
SIAM J. Computing 4(4):397-411, Dec., 1975.

[59] J.N. Gray et al.
Granularity of Locks and Degrees of Consistency in a Relational Database.
Technical Report, IBM San Jose, RJ 1654, Sept., 1975.

[60] Guttag, J., Horowitz, E. and Musser, D.
Abstract data types and software validation.
Comm. ACM 21(12):1048-1064, 1978.

[61] V.B. Gylys.
Optimal Partitioning of Workload for Distributed Systems.
In *Proceedings of the Annual Computer Conference, COMPCON*, pages 353-357. IEEE, Oct., 1976.

[62] Nico Habermann, Dewaynne Perry, Peter Feiler, Raul Medina-Mora,David Notkin, Gail Kaiser, Barbara Denny.
A Compendium of Gandalf Documentation.
Technical Report, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[63] A. Nico Habermann and Dewayne E. Perry.
Well-Formed System Compositions.
Technical Report, Technical Report, Department of Computer Science, Carnegie-Mellon University, March, 1980.

[64] F.E. Heart et al.
The Interface Message Processor for the ARPA Computer Network.
In *Proceedings SJCC, Vol. 36.* AFIPS, 1970.

[65] F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker.
A new Minicomputer/Multiprocessor for the ARPA Computer Network.
In *American Federation of Information Processing Societies, Proc. NCC, Vol.42.* 1973.

[66] Friedrich Hertweck, Eckart Raubold, Fritz Vogt.
X25 Based Process-Process Communication.
Computer Networks, North-Holland Publishing Company 2:250-270, Feb., 1978.

[67] P. Hibbard, A. Hisgen, and T. Rodeheffer.
A Language Design for a Multiprocessor Computing System.
In *Proceedings of the 5th Annual Symposium on Computer Architecture, Palo Alto, CA.*, pages 66-72. IEEE, Apr., 1978.

[68] C.A.R. Hoare.
Communicating Sequential Processes.
Comm. of the Assoc. Comput. Mach. 21(8), Aug., 1978.

[69] M. Hofri and C.J. Jenny.
On the Allocation of Processes in Distributed Computing Systems.
Technical Report, IBM Zurich Research Laboratory, Oct., 1978.

[70] R.R. Horsley and W.C. Lynch.
Pilot: A Software Engineering Case Study.
In *Proceedings of the Fourth International Conference on Software Engineering*, pages 94-99.
IEEE,ACM,GI, Sept., 1979.

[71] Ichbiah, J.D. et al.
Rationale for the Design of the ADA Programming Language.
ACM SigPlan Notices 14(6), June, 1979.

[72] J. D. Ichbiah, et al.
Rationale for the Design of the ADA Programming Language.
ACM SIGPLAN Notices 14(6B), June, 1979.

[73] Keki B. Irani and Nicholas G. Khabbaz.
A Model for Combined File Allocation for Distributed Data Bases.
In *First International Conference on Distributed Computing Systems, Huntsville, Alabama*, pages 15-21. IEEE, Oct., 1979.

[74] E.D. Jensen and G.A. Anderson.
Computer Interconnection Structures: Taxonomoy, Characteristics, and Examples.
Computing Surveys 7,4:197-213, Dec., 1977.

[75] E.D. Jensen.
The Archons Project.
Nov., 1981.
Notes of a talk at Carnegie-Mellon University.

[76] G. Jomier.
A Mathematical Model for the Comparison of Static and Dynamic Memory Allocation in Paged Systems.
IEEE Trans. Soft. Eng. SE-7(4):375-385, July, 1981.

[77] A. K. Jones and P. Feiler.
Software configuration experiments.
1979.
To appear.

[78] A.K. Jones, R.J. Chansler, I. Durham, P. Feiler, and K. Schwans.
Software Management of Cm* : A Distributed Multiprocessor.
In *NCC Conf. Proceedings*. NCC Conf. Proceedings, Assoc. Comput. Mach., 1977.

[79] A.K. Jones, R.J. Chansler, I. Durham, P. Feiler, D. Scelza, K. Schwans, and S. Vegdahl.
Programming Issues Raised by a Multiprocessor.
Proc. of the IEEE 66(2), Feb., 1978.

[80] A.K. Jones, R.J. Chansler, I. Durham, J. Mohan, K. Schwans, and S. Vegdahl.
StarOS, a Multiprocessor Operating System.
In *Proceedings of the Seventh Symposium on Operating System Principles, Asimolar, CA.* Assoc. Comput. Mach., Dec.10-12, 1979.

[81] Anita K. Jones and Karsten Schwans.
TASK Forces: Distributed Software for Solving Problems of Substantial Size.
In *Proceedings of the 4th Internatl. Conf. on Software Eng., Munich, W. Germany.* Assoc. Comput. Mach., GI, IEEE, Sept.14-16, 1979.

[82] Anita K. Jones and Karsten Schwans.
The TASK Specification Language.
1980.
Internal documentation of the StarOS project at Carnegie-Mellon University.

[83] Anita K. Jones and Karsten Schwans.
Specifying Software Configurations for Distributed Computation.
1982.
To be published.

[84] Anita K. Jones and Peter Schwarz.
Experience Using Multiprocessor Systems: A Status Report.
Surveys of the Assoc. Comput. Mach. 12(2):121-166, June, 1980.

[85] D.G. Kafura and V.Y. Shen.
Task Scheduling on a Multiprocessor System with Independent Memories.
SIAM J. Computing 6(1):167-187, March, 1977.

[86] Leonard Kleinrock.
Resource Allocation in Computer Systems and Computer-Communication Networks.
In *Proceedings International Conference Information Processing, North-Holland Publishing Co.*. IFIP, 1974.

[87] R.Kober.
The Multiprocessor System SMS 201 - Combining 128 Microprocessors to a powerful computer.
In *CompCon Fall 77*. Assoc. Comput. Mach., 1977.

[88] D.J. Kuck.
Illiac IV Software and Application Programming.
IEEE Trans. on Computing C-17(8):758-770, Aug., 1968.

[89] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe.
Dependence Graphs and Compiler Optimization.
In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia*, pages 207-218. Assoc. Comput. Mach., Jan., 1981.

[90] H.T. Kung.
Special-Purpose Devices for Signal and Image Processing: An Opportunity in VLSI.
Technical Report, Department of Computer Science, Carnegie-Mellon University, CMU-CS-80-132, July, 1980.

[91] H.T. Kung and D. Stevenson.
A Software Technique for Reducing the Routing Time on a Parallel Computer with a fixed Interconnection Network.
1977.
Unpublished working paper.

[92] David Alex Lamb.
Construction of a Peephole Optimizer.
Technical Report, Department of Computer Science, Carnegie-Mellon University, Aug., 1980.

[93] Butler W. Lampson.
Atomic Transactions.
In *In Butler W. Lampson, editor, Distributed Systems: Architecture and Implementation, an Advanced Course, chapter 14*. Springer-Verlag, 1981.

[94] Dorothy E. Lang.
Conference Report: Seventh International Symposium on Computer Architecture.
IEEE Computer Magazine 13(10):107-110, Oct., 1980.

[95] Anthony Lavia and Eric G. Manning.
Perturbation Techniques for Topological Optimization of Computer Networks.
In *Proceedings of the 4th Data Communications Symposium, Quebec*, pages 4-16-4-23. IEEE, Oct., 1975.

[96] Gerard LeLann.
A Distributed System for Real-Time Transaction Processing.
IEEE Computer Magazine 14(2):42-50, Feb., 1981.

[97] V. Lesser, D. Serain, J. Bonar.
PCL : A Process Oriented Job Control Language.
In *The 1rst Internatl. Conf. on Distributed Computing Systems, Huntsville, Alabama*, pages 315-329. Oct.1-5, 1979.

[98] V.R. Lesser and L.D. Erman.
An Experiment in Distributed Interpretation.
In *First International Conference on Distributed Computing Systems, Huntsville, Alabama.* Oct., 1979.

[99] Bruce W. Leverett.
Register Allocation in Optimizing Compilers.
Technical Report, Department of Computer Science, Carnegie-Mellon University, Feb., 1981.

[100] Art Lew.
Optimal Resource Allocation and Scheduling among Parallel Processes.
In *Proceedings of the Conference on Parallel Processing*, pages 187-202. Assoc. Comput. Mach., IEEE, 1975.

[101] Bernhard Lint and Tilak Agerwala.
Communication Issues in the Design and Analysis of Parallel Algorithms.
IEEE Trans. Soft. Eng. SE-7(2):174-188, March, 1981.

[102] Liskov, B.
Primitives for Distributed Computing.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 33-42. Assoc. Comput. Mach., Dec., 1979.

[103] Edward F. Gehringer and Robert J. Chansler, Mike Kazar.
StarOS User and System Structure Manual, Loader and Process Creator.
Technical Report, Department of Computer Science, Carnegie-Mellon University, June, 1981.

[104] Thelma Louie.
Array Processors: A Selected Bibliography.
IEEE Computer Magazine 14(9):53-59, Sept., 1981.

[105] James R. Low.
Automatic Coding: Choice of Data Structures.
Interdisciplinary Systems Research, ISR 16, Birkhaeuser Verlag, Basel, 1976.

[106] Bruce T. Lowerre.
The HARPY Speech Recognition System.
Technical Report, Department of Computer Science, Carnegie-Mellon University, April, 1976.

[107] Samy Mahmoud and J. S. Riordon.
Optimal Allocation of Resources in Distributed Information Systems.
Trans. on Database Systems 1(1):66-78, March, 1976.

[108] S.A. Mamrak, William A. Hagen.
Performance Measurement in Desperanto.
In *Third International Conference on Distributed Computing Systems, Miami, Florida*. Assoc. Comput. Mach, IEEE, Oct., 1982.

[109] Horst Mauersberg.
The Spice Debugger.
1981.
Working Paper of the DSN/SPICE Group at Carnegie-Mellon University.

[110] Stephen McConnel and Dan Siewiorek.
C.vmp: The Implementation, Performance, and Reliability of a Fault Tolerant Multiprocessor.
Technical Report, Department of Computer Science, Carnegie-Mellon University, March, 1978.

[111] Patrick F. McGehearty.
Performance Evaluation of a Multiprocessor under Interactive Workloads.
PhD thesis, Carnegie-MellonUniversity, Aug., 1980.

[112] P. McGregor and D. Shen.
Locating Concentration Points in Data Communication Networking.
In *Proceedings of the 4th Data Communications Symposium, Quebec*, pages 4-1-4-8. IEEE, Oct., 1975.

[113] R. Medina-Mora and Peter Feiler.
An Incremental Programming Environment.
IEEE Trans. Soft. Eng. SE-7(5):472-481, Sept., 1981.

[114] Leon J. Mekly and Stephen S. Yau.
Software Design Representation Using Abstract Process Networks.
IEEE Trans. Soft. Eng. SE-7(5):420-434, Sept., 1981.

[115] Robert M. Metcalfe, David R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Technical Report, CSL-75-7, Xerox Palo Alto Research Center, Palo Alto, CA., Nov., 1975.

[116] James G. Mitchell, William Maybury, and Richard Sweet.
Mesa Language Manual.
Xerox Palo Alto Research Center, 1978.

[117] Karsten Schwans.
The StarOS Module Loader.
1978.
Internal Working Paper of the StarOS Group, Computer Science Dept., Carnegie-Mellon Univ.

[118] H.L. Morgan and K.D. Levin.
Optimizing distributed data bases--A framework for research.
In *Proceedings of the National Computer Conference*. 1975.

[119] K. D. Levin and H. L. Morgan.
Optimal Program and Data Locations in Computer Networks.
Comm. Assoc. Comput. Mach. 20(5), May, 1977.

[120] Jim Morris.
The Cedar Project.
March, 1980.
NotesofatalkatCarnegie-MellonUniversity".

[121] Needham, R.M. and Walker, R.D.M.
Protection and process management in the CAP computer.
In *Proc. IRIA Internat'l Workshop on Protection in Operating Systems*, pages 155-160. Institute de Recherche d'Informatique et d'Automatique, France, Rocquencourt, France, 1974.

[122] Bruce Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, June, 1981.

[123] Edward F. Gehringer and Robert J. Chansler, Ivor Durham.
StarOS User and System Structure Manual, Object Management.
Technical Report, Department of Computer Science, Carnegie-Mellon University, June, 1981.

[124] P. Oleinick.
The implementation and evaluation of parallel algorithms on a multiprocessor.
PhD thesis, Carnegie-Mellon University, 1978.

[125] Organick, Elliot I.
,The Multics system: an examination of its structure.
The MIT Press, Cambridge, Massachusetts, 1972.

[126] S.M. Ornstein et al.
The Terminal IMP for the ARPA Computer Network.
In *Proceedings SJCC, Vol. 36*. AFIPS, 1970.

[127] Leon Osterweil.
Software Environment Research: Directions for the Next Five Years.
IEEE Computer Magazine 14(4):35-44, April, 1981.

[128] John K. Ousterhout, Donald A. Scelza, and Pradeep Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
Comm. of the Assoc. Comput. Mach. 23(2):92-104, Feb., 1980.

[129] John K. Ousterhout.
Partioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, April, 1980.

[130] D.L. Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Comm. of the Assoc. Comput. Mach. 15(12), 1972.

[131] D.L. Parnas.
On the Design and Development of Program Families.
In *Proceedings of the 2nd International Conference on Software Engineering, Oct. 1976, also in: IEEE Transactions on Software Engineering, Dec. 1976.* IEEE, Oct., 1976.

[132] C.V. Ramamoorthy and K.M. Chandy.
Optimization of Memory Hierarchies in Multiprogrammed Systems.
Journal of the Assoc. Comput. Mach. 17(3):426-445, July, 1970.

[133] C.V. Ramamoorthy and Benjamin W. Wah.
The Placements of Relations on a Distributed Relational Data Base.
In *Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama*, pages 642-650. IEEE, Oct., 1979.

[134] Richard F. Rashid.
Accent: A communication oriented network operating system kernel.
In *Proceedings of the Eighth Symposium on Operating System Principles, Asimolar, CA.* Assoc. Comput. Mach., Dec., 1981.

[135] Levy Raskin.
Performance Evaluation of Multiple Processor Systems.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, Aug., 1978.

[136] D. Raj Reddy and A. Nico Habermann.
A Proposal for Research on Signal Understanding in Distributed Systems.
July, 1979.
Working Paper of the DSN/SPICE Group at Carnegie-Mellon University.

[137] W.E. Riddle, J.H. Sayler, A.R. Segal, A.M. Stavely, J.C. Wileden.
A Description Scheme to Aid the Design of Collections of Concurrent Processes.
In *CompCon 1978*. Assoc. Comput. Mach., 1978.

[138] Edward F. Gehringer and Robert J. Chansler, Joe Mohan.
StarOS User and System Structure Manual, The Schedulers and Multiplexor.
Technical Report, Department of Computer Science, Carnegie-Mellon University, June, 1981.

[139] Patricia Griffiths-Selinger.
A Heuristic Algorithm for Routing Queries in a Dsitributed Data Base.
1980.
A Heuristic Algorithm for Routing Queries in a Distributed Data Base.

[140] Mary Shaw.
A Formal System for Specifying and Verifying Program Performance.
Technical Report, Technical Report, Department of Computer Science, Carnegie-Mellon University, June, 1979.

[141] D. Siewiorek and Vittal Kini, Eds.
Reliability in Multiprocessor Systems: A Case Study of C.mmp, Cm*, and C.vmp.
IEEE Trans. on Computing C-27, Oct., 1978.

[142] Ajay Singh.
Pegasus: A Controllable, Interactive Workload Generator for Multiprocessors.
Technical Report, Carnegie-Mellon University, Pittsburgh, Pa. 15213, Dec., 1981.
Master's Thesis, Electrical Engineering, Carnegie-Mellon University, CMU-CS-82-103.

[143] Edward F. Gehringer and Robert J. Chansler.
StarOS User and System Structure Manual, StarOS Six12.
Technical Report, Department of Computer Science, Carnegie-Mellon University, June, 1981.

[144] Richard Snodgrass.
Monitoring Distributed Systems.
PhD thesis, Computer Science Dept., Carnegie-Mellon Univ., Sept., 1982.

[145] R. Snodgrass.
An Object-Oriented Command Language.
IEEE Trans. Softw. Eng. , 1982.
To be published.

[146] Gary H. Sockut and Robert P. Goldberg.
Database Reorganization-Principles and Practice.
Journal of the Assoc. Comput. Mach. 11(4):371-396, Dec., 1979.

[147] M.H. Solomon and R.A. Finkel.
Roscoe -- A multiminicomputer Operating System.
Technical Report, TechReport321, Computer Science Department, University of Madison-Wisconsin, Sept., 1978.

[148] Computer Science Department, Carnegie-Mellon University.
Proposal for a Joint Effort in Personal Scientific Computing.
Technical Report, Computer Science Department, Carnegie-Mellon University, Aug., 1979.

[149] Edward F. Gehringer and Robert J. Chansler.
StarOS User and System Structure Manual.
Technical Report, Department of Computer Science, Carnegie-Mellon University, June, 1981.

[150] Harold S. Stone.
Multiprocessor Scheduling with the Aid of Network Flow Algorithms.
IEEE Trans. Softw. Eng. SE-3(1):85-93, Jan., 1977.

[151] G.S. Rao, H.S. Stone, and T.C. Hu.
Assignment of Tasks in a Distributed System with Limited Memory.
IEEE Trans. on Software Eng. C-28,4:291-299, April, 1979.

[152] Michael Stonebraker and Eric Neuhold.
A Distributed Data Base Version of INGRES.
Technical Report, Memorandum No. ERL-M612, Electronics Research Laboratory, University of California, Berkeley, Sept., 1976.

[153] R.J. Swan.
The Switching Structure and Addressing Architecture of an Extensible Multiprocessor : Cm.*
PhD thesis, Carnegie-Mellon Univ., 1978.

[154] Karsten Schwans.
The TASK Compiler.
1981.
Internal Working Paper of the StarOS Group, Computer Science Dept., Carnegie-Mellon Univ.

[155] K. J. Thurber and Leon D. Wald.
Associative Parallel Processors.
Surveys of the Assoc. of Comput. Mach. 7(4), Dec., 1975.

[156] Walter F. Tichy.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, Jan., 1980.

[157] K.H. Timmesfeld et al.
Pearl: A Proposal for a Process and Experiment Automation Real-Time Language.
Technical Report, Gesellschaft fuer Kernforschung mbH, Karlsruhe, PDV-Bericht KFK-PDV1, April, 1973.

[158] Kishor S. Trivedi, Robert E. Kinicki.
A Model for Computer Configuration Design.
IEEE Computer Magazine 13(4):47-57, April, 1980.

[159] Connie Umland Smith.
The Prediction and Evaluation of the Performance of Software from Extended Design Specifications.
PhD thesis, The University of Texas at Austin, Department of Computer Sciences, Aug., 1980.

[160] Robert S. Wilkov.
Analysis and Design of Reliable Computer Networks.
IEEE Trans. Comm. COM-20(3), June, 1972.

[161] Wm.A. Wulf et al.
Bliss : A Language for System Programming.
Comm. of the Assoc. Comput. Mach. 17, 1974.

[162] Wm.A. Wulf, E.Cohen, W.Corwin, A.K. Jones, C.Pierson, F.Pollack.
Hydra: The Kernel of a Multiprocessor Operating System.
Comm. of the Assoc. Comput. Mach. 17(6), June, 1974.

[163] Wm.A. Wulf, R.Levin, C.Pierson.
Overview of the Hydra Operating System.
In *Proceedings of the Fifth Symposium on Operating System Principles*. Assoc. Comput. Mach., SigOps, Nov., 1975.

[164] Wm.A. Wulf, Ralph L. London and Mary Shaw.
Abstraction and Verification in Alphard, Introduction to Language and Methodology.
Technical Report, Comp. Science Dept., Carnegie-Mellon Univ., June, 1976.

[165] Bruce Leverett, Roderic Cattell, Stephen Hobbs, Joseph Newcomer, Andrew Reiner, Bruce Schatz, William Wulf.
An Overview of the Production Quality Compiler-Compiler Project.
Technical Report, Department of Computer Science, Carnegie-Mellon University, Feb., 1979.

[166] William A. Wulf, Roy Levin, Samuel R. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill Advanced Computer Science Series, 1981.

[167] W.A. Wulf and C.G. Bell.
C.mmp-a multi mini processor.
In *AFIPS Conference Proceedings, Vol41, Part II, FJCC*. AFIPS, 1972.

[168] Andrew Chi-Chih Yao.
Scheduling Unit-Time Tasks with Limited Resources.
In *Proceedings of the Conference on Parallel Processing*, pages 17-35. Assoc. Comput. Mach., IEEE, 1975.

[169] Stephen S. Yau and Chen-Chau Yang.
An Approach to Distributed Comnputing System Software Design.
In *Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama*, pages 31-42. IEEE, Oct., 1979.